Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Network Attack Reference Data Set

J. McKenna and J. Treurniet

Defence R&D Canada – Ottawa
TECHNICAL MEMORANDUM
DRDC Ottawa TM 2004-242
December 2004

Canada

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 2004		**00-12-2004 to 00-12-2004**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Network Attack Reference Data Set (U)	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defence R&D Canada - Ottawa,3701 Carling Avenue,Ottawa, Ontario,CA,K1A 0Z4**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

A set of network attacks was created at DRDC Ottawa for the purpose of testing network attack detection and visualisation methods. The network attack traces were generated by extracting attacks from real-world networks, from closed networks specifically set up to test attacks, and through the use of custom software written to simulate attack traffic. In this document, the attacks included in the data set are described in detail along with the method used to generate them. The software tools used in the creation of the data sets are presented and issues involved in the generation of the data are discussed. The 52 attack traces are available on a CD in a purified form.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**78**	

Network Attack Reference Data Set

J. McKenna and J. Treurniet

Defence R&D Canada – Ottawa

Technical Memorandum

DRDC Ottawa TM 2004-242

December 2004

Abstract

A set of network attacks was created at DRDC Ottawa for the purpose of testing network attack detection and visualisation methods. The network attack traces were generated by extracting attacks from real-world networks, from closed networks specifically set up to test attacks, and through the use of custom software written to simulate attack traffic. In this document, the attacks included in the data set are described in detail along with the method used to generate them. The software tools used in the creation of the data sets are presented and issues involved in the generation of the data are discussed. The 52 attack traces are available on a CD in a purified form.

Résumé

On a créé une série de simulations d'attaques réseau à RDDC, à Ottawa, afin de mettre à l'essai les méthodes de détection et de visualisation des attaques sur le réseau. On a généré les traces laissées par les attaques réseau en procédant à l'extraction des données sur les attaques dans les réseaux du monde réel, les réseaux fermés configurés spécifiquement pour simuler les attaques ainsi qu'à l'aide de logiciels personnalisés créés pour simuler le trafic réseau lors des attaques. Vous trouverez dans le présent document la description détaillée des attaques définies dans les ensembles de données ainsi que des méthodes utilisées pour les générer. On y décrit également les outils logiciels utilisés pour créer les ensembles de données et les questions concernant la production des données. Les traces laissées par les 52 attaques sont disponibles sur CD sous forme épurée.

This page intentionally left blank.

Executive summary

Background

The Network Information Operations (NIO) section at DRDC Ottawa has identified network attack detection as a vital research area in the field of NIO. There is a requirement for further research in the area of Intrusion Detection (ID), as current techniques are thought to be unsatisfactory in some regard or another. Signature-based ID systems are rigid and lack the ability to change without human intervention, while anomaly detection techniques often result in a high degree of false positives and can sometimes be re-trained.

To verify their effectiveness and completeness, new methods for network intrusion detection and network attack traffic visualisation must be tested with actual network traffic. In the past this has largely been done with an arbitrary traffic set, or one which has not been completely investigated. Researchers require a set of known, well documented data with which to test their algorithms. This is the purpose of this body of work.

Principal Results

Attack traces were collected in four categories of attack: reconnaissance, escalation, denial of service and covert data transfer. The primary source of real traces was the Defence Research Establishment network (DREnet), which provided a large amount of reconnaissance traffic. Other required traces were either crafted using custom software or generated in a laboratory environment. One trace was extracted from the DEFCON 8 “Capture the Flag” contest traffic. In total, 52 traces were collected, along with one steganography example.

The IP addresses of all of the traces were altered so that the traces would appear as though they were extracted from the same network. The traces are presented in two formats on the distribution CD: the first includes the trace alone, and the second is the attack traffic placed in the middle of one hour of ambient traffic collected from the DREnet. This hour of traffic was first cleaned by removing all TCP anomalies and removing all suspicious traffic as detected by the Snort IDS. For the distribution intended for external parties, the ambient traffic was also purified, obfuscating all DREnet IP addresses and removing all packet payload.

Significance of Results and Future Work

This data set will enable the Attack Detection and Analysis group to test the limits of intrusion detection systems and to systematically test new intrusion detection techniques and network traffic visualisation techniques. The reference data set should be continually updated with new attacks. The capabilities of the software have been included in DRDC’s Network Traffic Analysis Toolbox as powerful packet and trace manipulation functions.

J. McKenna and J. Treurniet; 2004; Network Attack Reference Data Set; DRDC Ottawa TM 2004-242; Defence R&D Canada – Ottawa.

Sommaire

Contexte

La section des Opérations d'information de réseau de RDDC, à Ottawa, a défini la détection des attaques réseau comme un secteur de recherche essentiel à ses opérations. Il apparaît nécessaire d'approfondir les recherches dans le secteur de la détection des intrusions, car les techniques actuelles sont jugées insatisfaisantes à certains égards. Les systèmes de détection des intrusions qui utilisent l'approche par signature sont rigides et ne permettent pas d'apporter des modifications sans intervention humaine, tandis que les techniques de détection des anomalies se traduisent souvent par un taux élevé de faux positifs et peuvent parfois être perfectionnées.

Pour s'assurer de l'efficacité et de l'intégralité des nouvelles méthodes de détection des intrusions sur le réseau et de visualisation du trafic réseau lors des attaques, il faut en faire l'essai dans le trafic réseau réel. Dans le passé, les essais ont été menés en grande partie dans des ensembles de données arbitraires sur le trafic ou avec des données n'ayant pas fait l'objet d'un examen complet. Or, les chercheurs ont besoin d'utiliser des ensembles de données connues et bien documentées pour tester leurs algorithmes, et c'est de cela dont il est question dans les présents travaux.

Résultats principaux

On a recueilli les traces laissées par les attaques pour les quatre catégories suivantes : reconnaissance, élévation de privilège, déni de service et transfert de données secrètes. Le réseau DRENnet (Centre de recherches pour la défense) constituait la principale source de traces réelles et a fourni une grande partie des données sur le trafic relatif à la reconnaissance. Les autres traces requises ont été soit produites à l'aide de logiciels personnalisés, soit générées dans un environnement d'essai en laboratoire. On a extrait une trace du trafic relatif au jeu "saisir le drapeau" à DEFCON 8. Au total, 52 traces ont été recueillies, de même qu'un exemple de stéganographie.

On a modifié les adresses IP de toutes les traces pour donner l'illusion que les traces ont été extraites du même réseau. Les traces sont présentées en deux formats sur le CD de distribution : le premier représente la trace seule, et le deuxième représente le trafic local tiré du réseau DREnet soumis au trafic de l'attaque pendant une heure. Ce trafic a d'abord été nettoyé et dépouillé de toute anomalie de type TCP ainsi que de tout trafic suspect détecté par Snort IDS. Afin de permettre la distribution du CD à des parties externes, on a également épuré le trafic local, en masquant toutes les adresses IP du réseau DREnet et en retirant tous les paquets de charge utile.

Portée des résultats et travaux futurs

Cet ensemble de données permettra au groupe de détection et d'analyse des attaques de mettre à l'épreuve les limites des systèmes de détection des intrusions et de tester systématiquement

les nouvelles techniques de détection des intrusions et de visualisation du trafic réseau. L'ensemble de données de référence fera l'objet d'une mise à jour continue et tiendra compte de toute nouvelle attaque. La boîte à outils d'analyse du trafic réseau de RDDC renferme les fonctionnalités du logiciel sous forme de puissantes fonctions de manipulation des paquets et des traces.

J. McKenna and J. Treurniet; 2004; Network Attack Reference Data Set; DRDC Ottawa TM 2004-242; R & D pour la défense Canada – Ottawa.

This page intentionally left blank.

Table of contents

List of tables

This page intentionally left blank.

1 Introduction

Network attack detection and analysis plays an important role in network security. The methods currently employed in Intrusion Detection Systems (IDS) are thought to be unsatisfactory in some regard or another [1, 2]. Misuse detection techniques, through systems that detect bad network behaviour based on signatures, lack the ability to change without human intervention and therefore can introduce false negatives in its reports. Anomaly detection techniques, through systems that detect bad behaviour based on profiling, are widely thought to be poor in their false positive reporting. These methods also may allow re-training by the skilled attacker to accept malicious behaviour as normal. Research in this area is essential to the protection of our network infrastructure. DRDC Ottawa is currently investigating new techniques for attack detection, including strict anomaly detection and the application of neural network techniques.

New methods for network intrusion detection and network attack traffic visualisation must be tested with actual network traffic to verify their effectiveness and completeness. In the past this has largely been done with an arbitrary traffic set, or one which has not been completely investigated. This leads to doubt regarding the results of the test. Researchers require a set of known, well documented data with which to test their algorithms. This is the purpose of this body of work.

MIT's DARPA Intrusion Detection Evaluation data [3] is a well-known existing data set, which includes traffic with documented attacks on a small simulated network. In these traces, the logs are collected daily and the attacks are interspersed at random times. We aim to create traces that last for one hour and contain just one attack, using a network that is substantially larger.

Note that the definition of *attack* will vary from person to person. In this document, the term is used to describe any suspicious activity that occurs on the network, including scans.

2 Types of Attacks

Attacks can be categorized in many ways. Dain and Cunningham [4] separate attacks into 5 broad categories: network discovery, host discovery, escalation, denial of service and covert data transfer. We use this scheme, combining network and host discovery into one category, labelled reconnaissance, due to the overlap between the two categories.[1] This section gives a brief description of the kinds of attacks that are included in the reference data set.

2.1 Reconnaissance

We define reconnaissance as the attempt to gain information about hosts on a network, the services they offer, or the versions of software components installed. This is generally the

[1] Note that there is some overlap among the other categories as well. An improved taxonomy is under investigation and is beyond the scope of this report.

first stage in an attack, which helps the attacker find potential targets, or gain information about a specific target.

2.1.1 Scans

We define a network scan as a set of related probes directed at one or more hosts on a network. Typically, a network scan is used to determine the existence of network nodes or devices, and possibly the services they offer. When the attacker attempts to access system services in order to determine which are available, the scan is called a “portscan”.

Network scans can be classified as having “horizontal”, “vertical”, “block” or “strobe” footprints [5, 6]. To visualise the origin of these terms, consider a Cartesian plot with IP address on the x-axis and port on the y-axis. If a network scan probes a contiguous range of IP addresses for a single port of interest, a plot of (IP address,port) would appear as a horizontal line. A network scan of probes directed to one host and a contiguous range of ports would appear as a vertical line. Scans of IP address ranges for ranges of ports would appear as blocks. Probes directed to a non-contiguous set of ports on one or more hosts would constitute a strobe scan.

The three most prevalent protocols used in scanning are TCP, UDP and ICMP [7]. Because TCP and UDP work at the transport layer of the TCP/IP protocol suite [8], they use ports to establish connections. A probe for a service on a particular port can be performed using the TCP protocol: a TCP packet with the SYN flag set is a connection request; once established, the connection can be closed gracefully with a TCP FIN packet or torn down with transmission of a TCP RST packet. A TCP packet with disallowed flag combinations, *e.g.* with both SYN and FIN flags set, can also be used to elicit a response from hosts on the network. Acceptable flag combinations can elicit a response when directed to a host that has not initiated a connection. At one time these were considered stealth techniques [9], but they are now easily detected. Inverse mapping can be accomplished by using TCP RST packets: if a host does not exist, some routers will happily respond to a TCP RST with an ICMP host unreachable message [9].

A probe for a service on a particular port may also be carried out using the UDP protocol. When an empty UDP datagram is sent to an open port, the host either responds with an error message or makes no response. If a UDP packet is sent to a closed port on a host, the host may respond with an ICMP port unreachable message. This allows an attacker to test for the presence of filtering.

The ICMP protocol, which operates at the network layer of the TCP/IP protocol suite, provides a means of testing for the existence of hosts on a network. If a host is up and there is no filtering for ICMP, an ICMP echo request (commonly termed a “ping”) will elicit an echo reply. Note that probes for a range of IP addresses using a protocol that does not use ports, such as ICMP, would be classified as a horizontal scan.

2.1.2 Traceroute

Traceroute is a tool used to map the route between one host and another, and may be used by an attacker to provide a map of the network of interest. It works on the premise that the time-to-live (TTL) field in the IP header is decremented at each router on it's path. When the TTL of a packet reaches 0, an ICMP time exceeded message is sent back to the originating host.

The process of tracing the route between two hosts proceeds as follows. A UDP or ICMP echo request packet is sent out, with an initial TTL value of 1. When it reaches the first router, an ICMP time exceeded message is returned. Packets are sent in this fashion, with the TTL value of each new packet incremented by 1, until the target host is reached and a complete map of the route between the two hosts is created.

2.1.3 Operating System Identification

Identification of an Operating System (OS) from its response to stimuli is called fingerprinting. Most OSs, and some OS versions, can be differentiated by these responses, based on the behaviour of their TCP or ICMP implementation. Common OS fingerprinting tools include *QueSO* [10] (literally translates to "what OS") and *nmap* [11], however there are a number of additional tools available for download.

2.1.4 Vulnerability Assessment Tools

Vulnerability Assessment (VA) tools are meant to be used by network security administrators to assess the security of their networks, but may be used by attackers to identify vulnerable hosts. The CyberCop tool [12] has been used for years and is still effective in identifying vulnerabilities in a system.

2.1.5 Stealth Methods

Stealth techniques are often employed to prevent detection of scans by IDS. Methods of avoiding detection include randomization of IP sequences, slowing the rate of probing, and introducing random time intervals between probes. These methods decrease the likelihood that an IDS will correlate the individual probes. An attacker may also randomize non-essential fields to avoid the presence of a "tool signature" [5].

In the "needle in a haystack" technique, an attacker launches a very noisy (fast and voluminous) scan from one host under its control, and during this scan sends a few probes from another host to the target. The attacker's actual target is obscured by the noise and the attack may thereby go unrecognized. Distributed or coordinated scanning [13], similar to distributed denial of service, uses a number of compromised hosts, each collaborating to scan different areas of a network or host. The identity of the attacker is hidden.

Stealth methods can be applied to any of the reconnaissance techniques described here.

2.2 Escalation

The escalation category [4] consists of attacks that attempt to gain access to the target. Examples of escalation attempts include buffer overflow attacks and password cracking.

2.2.1 Buffer Overflow

Buffer overflow attacks can allow an attacker to run arbitrary code on a host. This is accomplished by sending more data to an application than the buffer was designed to handle. The extra data overflows into an adjacent buffer, which may result in the execution of code.

2.2.2 Password Cracking

Password cracking can be done using online methods or offline methods. Online methods use a list of common passwords to try to guess the password. Repeated connection attempts are made using the passwords in the dictionary, which makes this method very noisy and not always successful. Software such as *John the Ripper* [14] is an online password guessing package that is intended for local use by security administrators, to disallow the use of easily-guessed passwords. In an attack using offline methods, a copy of the `/etc/passwd` and `/etc/shadow` files are retrieved and attempts to guess the passwords are made locally. The *Brutus* [15] tool combines both of the above methods of password cracking, and is also intended for use by security administrators.

2.3 Denial of Service

Denial of Service (DoS) attacks compromise the availability of a host. This can be accomplished by exploiting a vulnerability (commonly with a single packet), effectively disabling a service or operating system, or by overflowing the host's capacity for traffic with an unmanageable amount of data (a "packet flood").

2.3.1 Vulnerability DoS

The *Land* attack [9] (named after an exploit program, `land.c`) is an example of a vulnerability DoS attack. The attack forges the source IP to be identical to the destination IP and thus creates a loop which can crash vulnerable operating systems. Although most modern operating systems are not vulnerable to this exploit, there are still several unpatched versions out there (notably Windows '95).

The *Teardrop* attack [9] has an effect similar to the *Land* attack. A fragmented packet is received and its data is written into memory. A second packet containing an incorrect offset that overlaps this memory is received, resulting a system crash. This has been patched on most operating systems, but some vulnerable hosts still remain in use (again, notably Windows '95).

2.3.2 Multipacket DoS

DoS can also be achieved through packet flooding. This can be done by overwhelming either the link to the host, or the host itself.

The *Smurf* attack [9] uses IP broadcast addresses to magnify traffic. It sends an ICMP echo request to several network broadcast addresses, with a return IP address spoofed to that of the victim. The resultant flood of ping responses overwhelms the link to the victim, resulting in a denial of service by packet flood. The ICMP protocol has been amended to specify that routers/gateways may include an option to drop traffic directed at IP broadcast addresses [16], however this option is not always implemented by networks [17].

Many older operating systems are vulnerable to a *TCP SYN flood* attack [9]. In this attack, TCP connection requests are received by the victim, and TCP puts each in queue for a response. While the maximum number of uncompleted (half-open) connections that may exist is exceeded, the victim is rendered unable to establish legitimate connections.

In the Distributed Denial of Service (DDoS) attack [18], an attacker compromises a host and installs a DDoS "master" program. From there, "slave" programs are installed on several, sometimes thousands, of compromised hosts. At the time of attack, the master simultaneously orders the slaves to begin a packet-flood DoS attack on the same victim.

Note that backscatter effects of multipacket TCP DoS can be mistaken for a reset scan. The victim of the DoS will send TCP reset packets to the originating IP addresses. If these addresses were spoofed to appear to be of the same address space, the activity will take on the appearance of a fast reset scan.

2.4 Covert Data Transfer

Covert data transfer includes methods of relaying information in a manner that is difficult to detect. Both tunnelling and information hiding are included in this category.

2.4.1 Tunnels

The ICMP protocol can be used to transfer data. The data is written in the content of ICMP echo request and reply packets, and may be encrypted. The *Loki* [19] tool is one well-known example.

2.4.2 Information Hiding

The information hiding category includes methods of hiding information in undetectable ways. For example, steganography transfers information through subtle changes in electronic pictures.

The header fields in TCP packets can be used to carry data [20]. The IP ID and TCP sequence number fields can carry ASCII character values. The TCP acknowledgement

number can also be used if a packet carrying information in it's sequence number is bounced to the forged IP of the true destination host.

3 Generation of Reference Attack Data

Traffic collected from the Defence Research Establishment network (DREnet) in August 2000, April 2001 and December 2002 provided a good source of attempted reconnaissance traces, but was lacking in variety. The DEFCON [21] annual security conference, which features a "Capture the Flag" contest in which participant groups attempt to secure their own host while attacking others on a local network, was expected to be a plentiful source of attack traces. Unfortunately, the data suffered badly from dropped packets and hence was a viable source for only one attack trace. The reference data set was completed by generating the necessary attacks either by using published exploits on a closed network or by crafting the attack trace using custom-built tools.

The reference data set is stored as a series of pcap format [22] files, which is generally accepted to be the standard format for archived traffic. The corresponding *tcpdump* tool creates and reads pcap files from network traffic. Other popular tools in the intrusion detection community support the pcap file format, including Ethereal [23], which may be used to translate pcap files to other formats.

The data set contains two versions of each attack: one version contains only the packets involved in the attack, and the other contains the attack packets merged with standard set of what we will call "normal" traffic, to be discussed further in Section 3.6.

To ensure consistency, all traces were altered to appear as if they had been sniffed on the same network. Three class B address ranges were required to accommodate the DREnet traces. The IP addresses of all target hosts were modified to have IP addresses in the range 172.16.0.0/16, 172.17.0.0/16 or 172.18.0.0/16. In closed network, DEFCON and crafted traces, further manipulation was necessary to create realistic traces. The Time-to-live (TTL) fields of the original attack packets were reduced to imply a greater number of hops between hosts. "Jitter", the variation in packet arrival time due to network delays, was also introduced. In crafted and closed network traces involving multiple packets, packet dropping was introduced at a rate of 0.5%–1% [24]. Such manipulation was performed using custom tools, described in Section 3.1. Note that after alteration of a packet, the IP and TCP/UDP checksums were recalculated as required.

3.1 Utilities

There are numerous tools [22, 25] available for analyzing or displaying data in pcap files. While tools exist to perform simple manipulations of pcap files, such as dividing or concatenating traces [26] and purifying traces through modifying IP address and removing content [27], there is a shortage of tools capable of performing complex manipulations of them, such as modification of header fields and timestamps, and the merging of interleaved

traffic data.

A library of shared functions, called *TCPLib*, was written to provide an interface to the packets in a pcap file. *TCPLib* does not rely on libpcap (Unix) or WinPcap (Windows) for effective operation. A series of tools collectively dubbed *TCPUtils* was created to allow for the manipulation and examination of pcap files. These tools are listed in Table 1.

Table 1: *The tools created for manipulation and analysis of pcap files. The tools are described in more detail in Appendix A.*

Tool name	Usage
`tcpiptranslate`	Used to replace instances of one IP address with another.
`tcpttltranslate`	Alters the IP TTL field in each IP packet by a specified amount.
`tcptimeshift`	Alters each packet time stamp by a specified amount, allowing a trace to appear as if it took place at a different time.
`tcpmerge`	Merges two or more pcap files which may have interleaved time stamps.
`tcpjitter`	Adds realism to traces by altering spacing between timestamps up to a specified percentage, or by dropping a specified percentage of packets.
`tcptimestretch`	Expands or compresses gaps in time between packets.
`tcptrunc`	Truncates packet captured data. This is useful when merging several traces, each of which may have different packet capture lengths.
`tcprate`	Creates a tab-delimited table of packet rates, in bytes per second.
`tcpcontent`	Displays packet payloads in ASCII.

3.2 DREnet Traffic

The Snort IDS [25] was used to analyze data captured on the DREnet. Snort is a signature-based IDS, based on a set of rules to identify known attacks. The default set of rules shipped with Snort 2.0 was used, as well as an updated list of rules for current attacks [28]. Alerts generated by Snort were manually investigated. When an attack was identified in the traffic, *tcpdump* was used to isolate the packets involved in the attack.

3.3 DEFCON Traffic

Exploits were identified within traces from the Capture the Flag contest at DEFCON 8 [29]. Since the DEFCON 8 data suffered badly from dropped packets, it was decided that the focus for this data would be to find a buffer overflow attack for which the trace was intact. The Snort IDS was used to identify such attacks and the data was searched for a complete trace. When such an attack was identified in the traffic, *tcpdump* was used to isolate the attack trace. IP addresses and TTL values were modified in the trace using *TCPUtils*, so that the trace was consistent with the other traces in the data set.

3.4 Closed Network Traffic

Tools to launch the some of the exploits used in the generation of attack traffic on a closed network were obtained from the Internet. The attacks were launched on a test network consisting of three machines, two Linux and one Windows, and the traffic was captured regardless of the success of the attack. IP addresses and TTL values were modified in the trace, and jitter and packet loss were introduced using *TCPUtils*.

3.5 Crafted Traffic

In some cases, especially those involving distributed attacks, simulating an attack with a small test network was not feasible. As a result, programs were written to generate pcap format files containing hypothetical traces for these types of attacks. The *TCPlib* library of functions was used to create the files. Using *TCPUtils*, the traces were crafted to use consistent IP addresses and realistic TTLs. Random packet dropping and time stamp fluctuations were also introduced.

3.6 Merging Isolated Traces with Clean Data

To ensure only one attack in each traffic trace, each isolated attack trace, regardless of its source, was merged with a one hour data set of ambient traffic representative of "normal" network activity.

In an attempt to create a clean data set, a strict anomaly detection algorithm [30] was applied to the TCP traffic in one hour of DREnet traffic. This algorithm detects illegal flag sequences and state transitions, and is successful in detecting scanning activity and other unusual behaviour. All anomalous TCP activity was removed from the traffic using a simple tcpdump filter. The resulting traffic was passed through the Snort IDS to verify clean content, and all suspicious traffic was again removed via a tcpdump filter. At this stage, the traffic is presumed to be "clean" (free of attacks) to the best of our knowledge. Using *TCPUtils*, the IP addresses of this traffic were translated to reflect the chosen class B address ranges 172.16.0.0/16, 172.17.0.0/16 or 172.18.0.0/16. The resulting data set is appropriate for internal use, however for a releasable distribution a second clean data set was generated where the class B addresses were obfuscated and the content removed from most packets using *TCPurify* [27]. The clean traffic is included in the data set as a baseline for "normal" activity.

Attacks that were less than one hour in length were merged with the clean traffic, shifted to approximately the half-hour mark. Attacks that ran longer than one hour were truncated in the merged version. An 1-hour window of data was selected in the attack traffic. For most files, this hour begins at 15000s into the attack trace and ends at 18600s into the attack trace. This hour was merged with the clean traffic. The `tcptimeshift` and `tcpmerge` utilities were used to merge each isolated attack with the clean traffic, with special attention paid to the packet capture length of the attack traffic.

4 Attacks Included in the Data Set

This section contains a description of each of the recorded attacks in the data set. The section is partitioned into the categories as they were presented in Section 2. When an attack type was not found in the data collected from the DREnet, it was generated on a closed network or crafted (excluding the buffer overflow that was obtained from DEFCON).

4.1 Reconnaissance

Recall that reconnaissance includes scans, OS fingerprinting, traceroute, the use of vulnerability assessment tools, and stealth activity. We include a section for traffic that may be interpreted as a scan but may also be a false positive, observed as a backscatter effect.

This section is organized as follows:

- Scans:
  - Fast horizontal scans
  - Fast strobe scan
  - Fast vertical scans
  - Fast block scans
- Stealth activity:
  - Slow horizontal scans
  - Slow strobe scans
  - Slow vertical scans
  - Needle in a haystack
  - Coordinated scans
- Traceroute
- OS identification
- Vulnerability assessment tools
- Possible false positives

Figures are included for some of the scans to illustrate interesting characteristics of the trace. The plot is a simple diagram of IP address versus time. The x-axis is time and the set of all unique IP addresses that were involved in the trace are equally spaced along the y-axis. Two points are rendered for each packet in the trace: one for the source IP address of the packet and one for the destination IP address.

4.1.1 Scans

Almost all scans located in the DREnet traffic were of the “horizontal” type, where a range of IP addresses is scanned for a single port. The “vertical” scan, where a single host is probed for a range of ports, had to be generated in the lab on a closed network. The “block” scan, which combines both horizontal and vertical scans, had to be crafted.

4.1.1.1 Fast Horizontal Scans

Filename: scan_H_ICMP_fast_seq.tcp
Source: DREnet, 8 December 2002
Duration: 4h 27m 11.7s
Description: A fast semi-sequential ICMP scan of 15663 hosts on a class B network. The 31275 packet scan is bursty: an echo request is sent to a group of 5-10 hosts, and re-tried about 2 seconds later. The process repeats every 2 seconds. Figure 1 shows the pattern. There is a 74 minute break about 2/3 of the way through the scan.

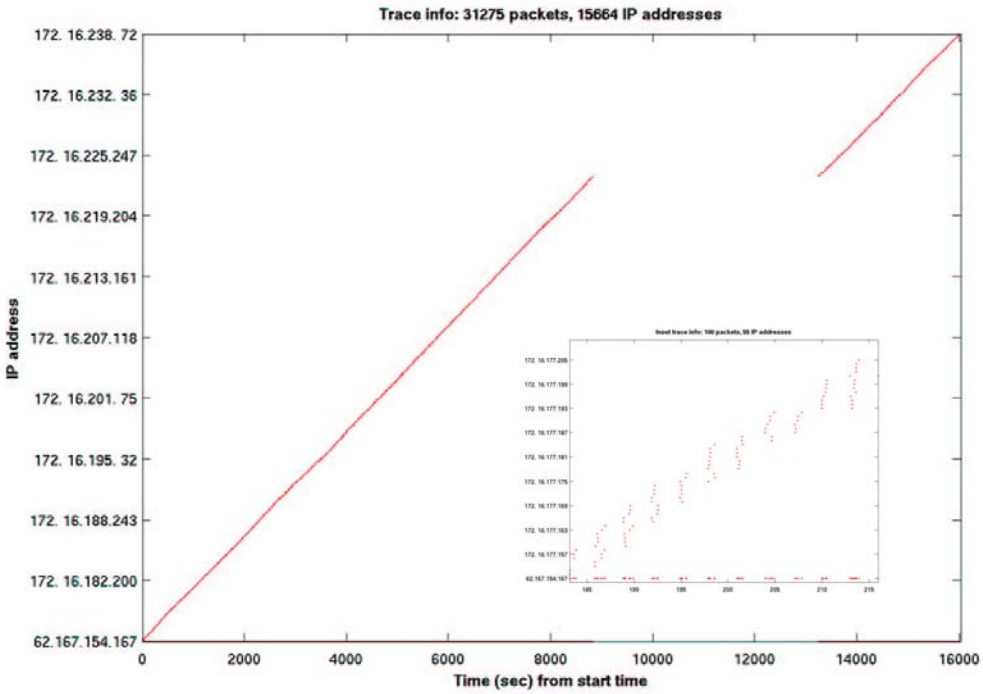

Figure 1: *Echo request scan. The inset shows the pattern of packet arrival with time.*

Filename: scan_H_UDP_fast_ran.tcp
Source: DREnet, 8 December 2002
Duration: 10m

Description: About 35 NetBIOS queries per second on port 137, from 185 unique sources probe 19669 unique destinations on 3 class B networks. 21227 packets are sent in total, however the destination hosts consist of 489 ranges of IP addresses (Figure 2). The captured activity occurs over a 10 minute period, but is representative of the day. This type of activity may indicate worm propagation through Windows shares.

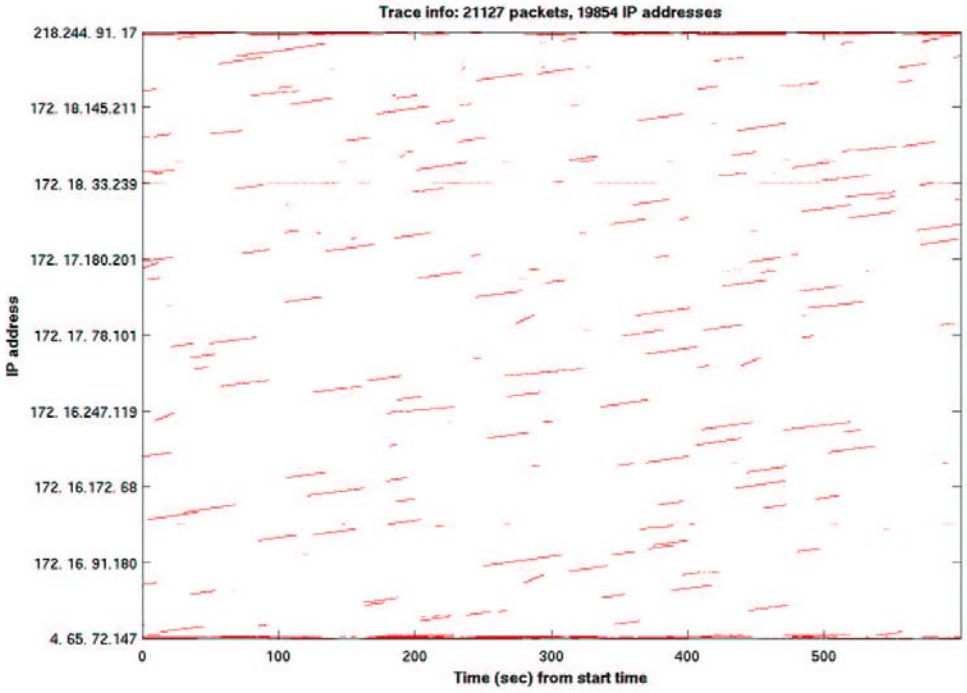

Figure 2: *A scan for Windows shares on port 137 (NetBIOS).*

Filename: scan_H_UDP_fast_seq.tcp
Source: DREnet, 29 April 2001
Duration: 5h 44m 47.4s
Description: A class B network is scanned sequentially for ports 407/UDP and 1419/UDP. 65493 unique destination hosts are scanned, covering almost all of the class B address space. The packets arrive at an average rate of 6.33 packets/second. The source port is constant for all probes.

Filename: scan_H_TCP_fast_TCPopts.tcp
Source: DREnet, 29 April 2001
Duration: 28m 59.2s
Description: A host makes 39343 TCP SYN connection attempts to port 53 on 24017 unique destination IP addresses at a rate of 22.6 packets/second. The scan

is not quite sequential, but also not quite random. Overall, the scan moves in the direction of increasing IP address, however there is some backtracking and some IP addresses are skipped. The inset of Figure 3 shows the pattern. The unusual feature of this scan is that TCP options are used.

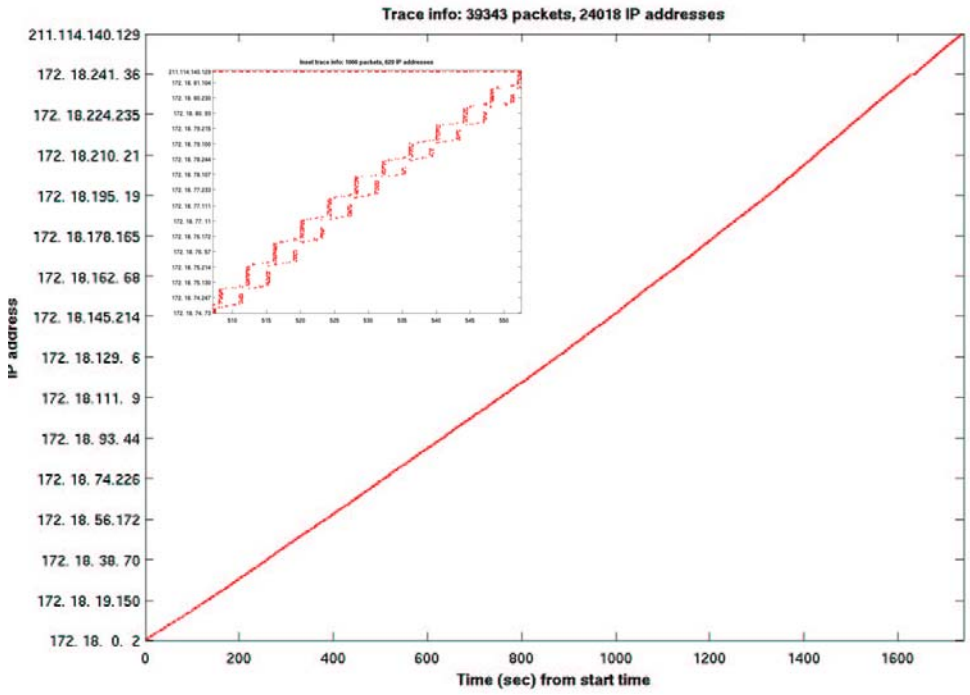

Figure 3: *Fast horizontal scan with backtracking.*

Filename: scan_H_TCP_fast_seq_skippy.tcp
Source: DREnet, 8 December 2002
Duration: 18h 7m 9.8s
Description: TCP SYN connection attempts to port 80. The IP addresses are generally increasing, but there are gaps in the class B address space that is being scanned, several of them being quite large gaps. There are 247958 packets in the trace, but just 27615 unique destination IP addresses. Each IP address is retried 9 times. The average rate of packet arrival is 3.80 packets per second. See Figure 4.

Filename: scan_H_TCP_fast_seq_classC.tcp
Source: DREnet, 8 December 2002
Duration: 2.1s

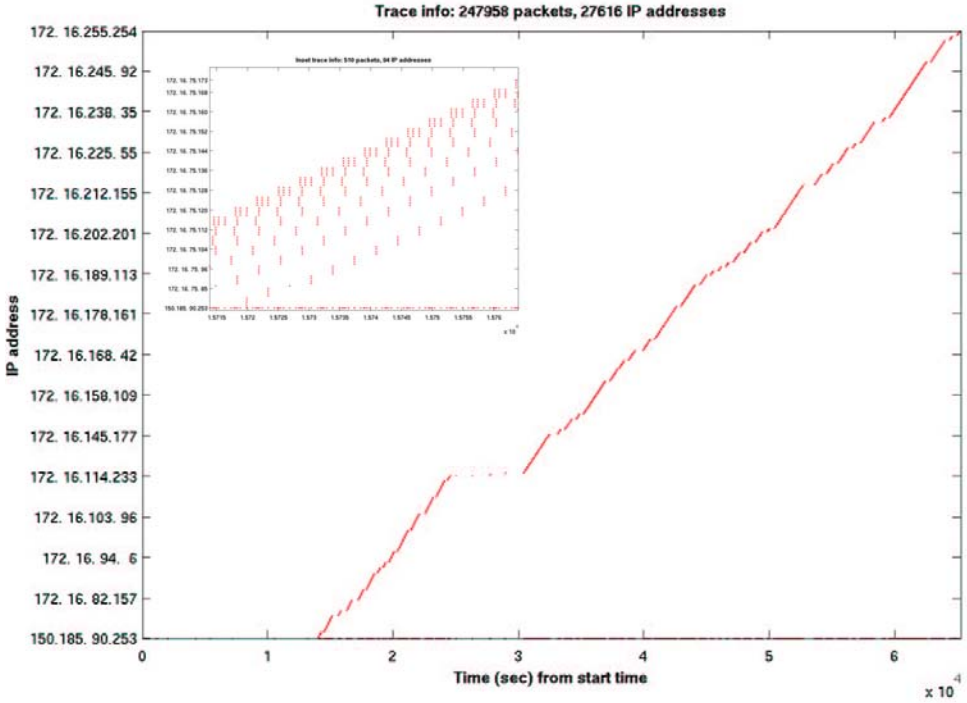

Figure 4: *Fast horizontal scan with gaps. The inset shows the pattern of TCP retries.*

Description: TCP SYN scan of an address space corresponding to a class C network (254 hosts) for a known trojan port. The packets arrive at an average rate of 120 packets/second, and there are gaps in the timing, as seen in Figure 5.

Filename: scan_H_TCP_fast_seq_3classB.tcp
Source: DREnet, 8 December 2002
Duration: 4h 19m 40.1s
Description: TCP SYN scan of three class B networks for one port (1433/sql). Although the scan appears to be sequential in the large-scale view, a close-up shows repetition (Figure 6). The trace contains 344405 packets, 30 of which are responses from internal hosts, at an average rate of 22.11 packets/second. There are 195480 unique destination IP addresses.

Filename: scan_H_TCP_fast_seq_synfin.tcp
Source: DREnet, 17 August 2000
Duration: 43.7s
Description: TCP SYN/FIN scan of three class B networks to TCP destination port 111 (sunrpc). The source port is also 111. Although sequential overall, investigation shows some variation in the order of arrival, mainly for the first 100

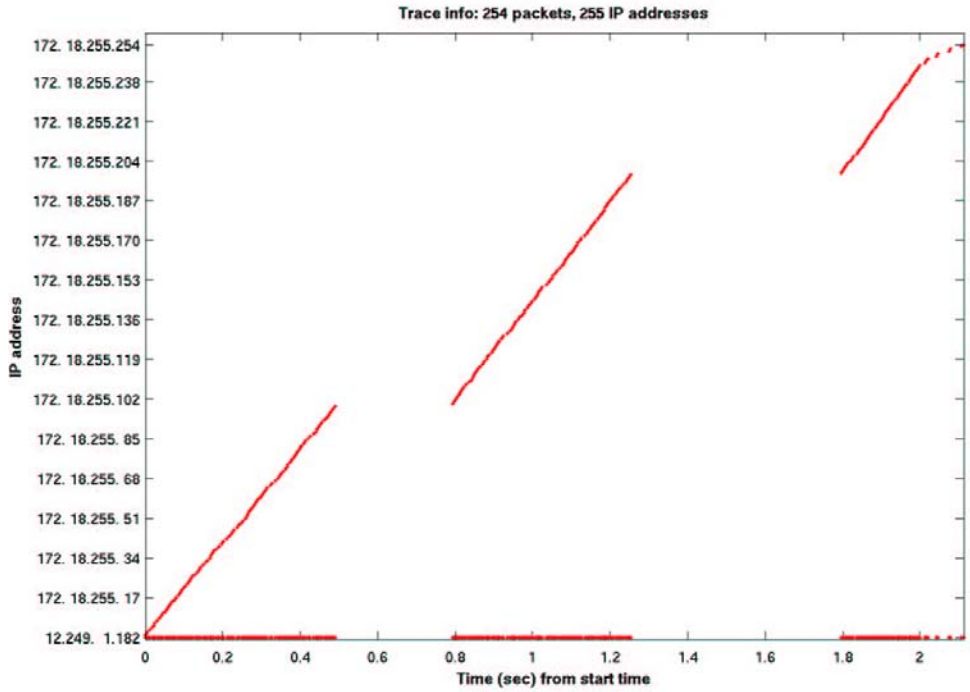

Figure 5: *Fast small network scan.*

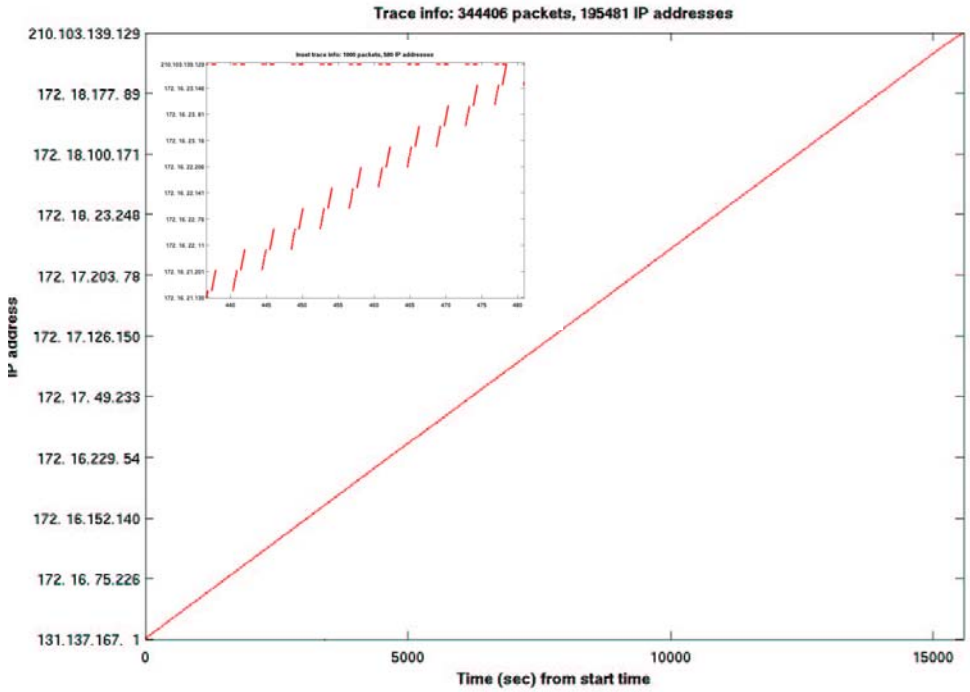

Figure 6: *Fast scan of 3 class B networks. The inset shows the repetition of attempts.*

packets. The trace contains 19721 packets targeting 19721 unique destination IP addresses, arriving at an average rate of 451.4 packets per second. There is, however, a pause in the scan lasting 17.2 seconds.

4.1.1.2 Fast Strobe Scan

Filename: scan_S_TCP_fast_ran.tcp
Source: DREnet, 29 April 2001
Duration: 1m 23.0s
Description: Fast TCP SYN connection attempts to 21 different ports on one host (often several times to each port), 12 of which are well-known back doors. The pattern is repeated on a different host 35 minutes later. The total duration of the trace including the scan of the second host is 38m 13.4s.

4.1.1.3 Fast Vertical Scans

The following portscans were crafted using `simPortScan.cc` (Appendix B.2).

Filename: scan_V_TCP_fast_seq_small.tcp
Source: Crafted (`simPortScan.cc`)
Duration: 51.1s
Description: A fast, ordered scan of ports 1-1024. The probes are sequential at a rate of 20 packets per second with 3% jitter, or having a delay between packets in the interval [0.0485s, 0.0515s].

Filename: scan_V_TCP_fast_ran_small.tcp
Source: Crafted (`simPortScan.cc`)
Duration: 51.1s
Description: A fast scan of ports 1-1024 in random order. The probes are randomized at a rate of 20 packets per second with 3% jitter, or having a delay between packets in the interval [0.0485s, 0.0515s].

Filename: scan_V_TCP_fast_seq_large.tcp
Source: Crafted (`simPortScan.cc`)
Duration: 54m 36.5s
Description: A fast, ordered scan of ports 1-65535. The probes are sequential at a rate of 20 packets per second with 3% jitter, or having a delay between packets in the interval [0.0485s, 0.0515s].

Filename: scan_V_TCP_fast_ran_large.tcp
Source: Crafted (`simPortScan.cc`)
Duration: 54m 36.7s

Description: A fast scan of ports 1-65535 in random order. The probes are randomized at a rate of 20 packets per second with 3% jitter, or having a delay between packets in the interval [0.0485s, 0.0515s].

4.1.1.4 Fast Block Scans

The block scans contain as many packets as can be theoretically fit on a 295kbps line, approximately 700 packets per second. The traces show only one hour of the block scan, where in reality they would span multiple hours.

Filename: scan_B_UDP_fast_seq_small.tcp
Source: Crafted (simBlockScanNormal.pl)
Description: A block scan, sequential in both IP address and port. Ports 1-1024 are probed for as many IP addresses as can be fit into the hour. No jitter was introduced.

Filename: scan_B_TCP_fast_seq_large.tcp
Source: Crafted (simBlockScanNormal.pl)
Description: A block scan, sequential in both IP address and port. Ports 1-65535 are probed for as many IP addresses as can be fit into the hour. No jitter was introduced.

Filename: scan_B_TCP_fast_ran_large.tcp
Source: Crafted (simBlockScanRandom.pl)
Description: A block scan, randomized in both IP address and port. A class C address space is chosen and the last octet is randomized along with the choice of port (1-65535). The jitter introduced in this trace is high, with delay between packets uniformly distributed within the range [5e-6s, 0.016s].

4.1.2 Stealth Activity

Recall that stealth activity includes slowing the rate of packet arrival, hiding one's identity in a large amount of decoy traffic, and using other compromised hosts to perform a scan.

A slow, randomized IP block scan is not included in the reference data due to the size of the required file. Distributed scans were also not present in the DREnet data and were crafted.

4.1.2.1 Slow Horizontal Scans

Filename: scan_H_TCP_slow_ran.tcp
Source: DREnet, 6 December 2002
Duration: 72h 39m 3.7s
Description: A very slow TCP SYN scan consisting of 456 packets to port 80 on 160 unique hosts residing on 3 class B subnets. The average time interval between probes is 9.56 minutes. The randomization of IP addresses is shown in Figure 7.

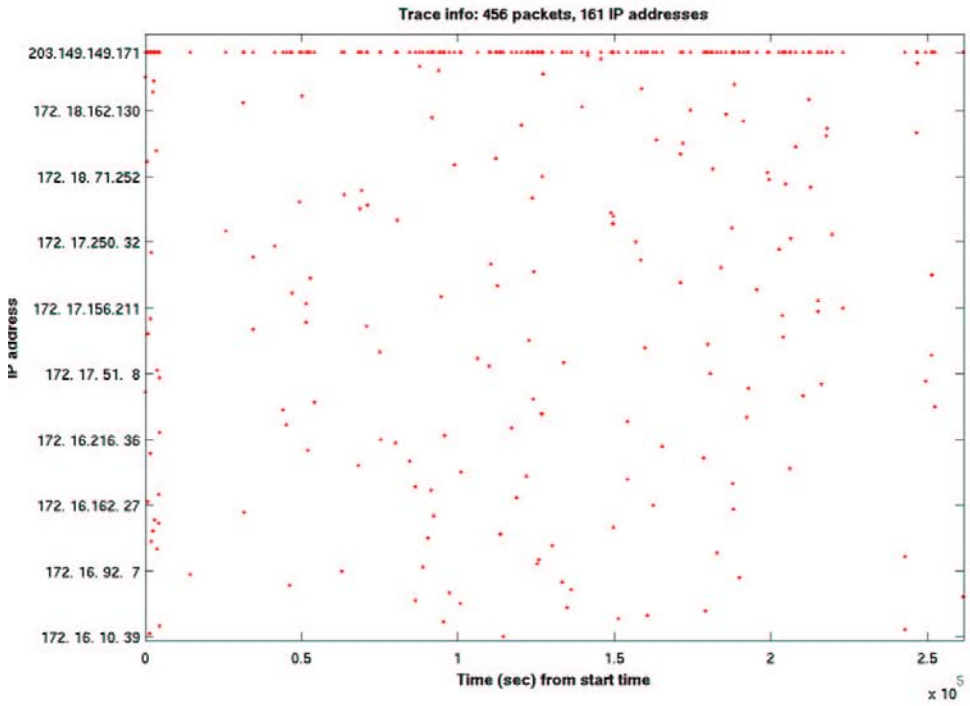

Figure 7: *Very slow, randomized scan of 3 class B networks. Each dot consists of 3 connection attempts.*

Filename: scan_H_TCP_slow_ran_toolsig.tcp
Source: DREnet, 29 April 2001
Duration: 9h 25m 27.8s
Description: A very slow TCP SYN scan consisting of 17 packets to port 515 on 17 unique hosts residing on 3 class B subnets. The order of the IP addresses appears to be randomized, as does the time interval between probes. The average time interval is 33.36 minutes. The attack tool used to perform the scan, *myscan* [31], is easily identifiable through a constant source port of 10101 and sequence number of 100.

Filename: scan_H_UDP_slow_seq.tcp
Source: Crafted (`simHPortScan.cc`)
Duration: 6d 17h 8m 41.1s
Description: A slow UDP scan to port 69 (tftp). 65023 packets are sent, with delays uniformly distributed within the interval [8.7s, 9.1s], to 65023 unique destination IP addresses. A packet drop rate of 0.4% was introduced.

Filename: scan_H_ICMPACK_slow_ran.tcp
Source: DREnet, 29 April 2001
Duration: 12h 56m 40.2s
Description: Slow ICMP and ACK combined scan. The probes come in pairs, with an echo request to the host followed by an ACK packet to port 80 originating from port 50990 or 50991. 11 hosts from 3 class B subnets are scanned in random order, with an average time interval between packets of 24.27 minutes.

4.1.2.2 Slow Strobe Scan

Filename: scan_S_TCP_slow_seq.tcp
Source: Crafted (simStrobeScan.pl)
Duration: 34h 6m 27.5s
Description: A slow TCP SYN scan is performed on a class C network. A set of packets aimed at 12 commonly-used services is sent with uniformly distributed delay between packets within the range [10s, 70s] (75% jitter with an average delay of 40s) to 254 unique destination IP addresses.

4.1.2.3 Slow Vertical Scans

Filename: scan_V_TCP_slow_seq.tcp
Source: Crafted (simPortScan.pl)
Duration: 34h 14m 32.3s
Description: A slow ordered scan of ports 1-1024 with with delay between probes uniformly distributed within the range [90s, 150s].

Filename: scan_V_TCP_slow_ran.tcp

Source: Crafted (simPortScan.pl)
Duration: 65h 36m 9.3s
Description: A very slow scan of ports 1-1024 in random order, over the course of 65 hours. The delay between packets is uniformly distributed within the range [57.5s, 402.5s].

4.1.2.4 Needle in a Haystack

Filename: scan_H_TCP_fast_seq_needle.tcp
Source: DREnet, 8 December 2002
Duration: 6m 15.9s
Description: A fast, sequential TCP SYN scan for ports 1243 and 27374 takes attention away from two other IP addresses probing the same ports. The large scan sends 12371 packets to 2439 unique hosts on one class B subnet at a rate of 32.9 packets per second. In the midst of this traffic, two other hosts send probes to two IP addresses, shown by the green and blue open circles in Figure 8.

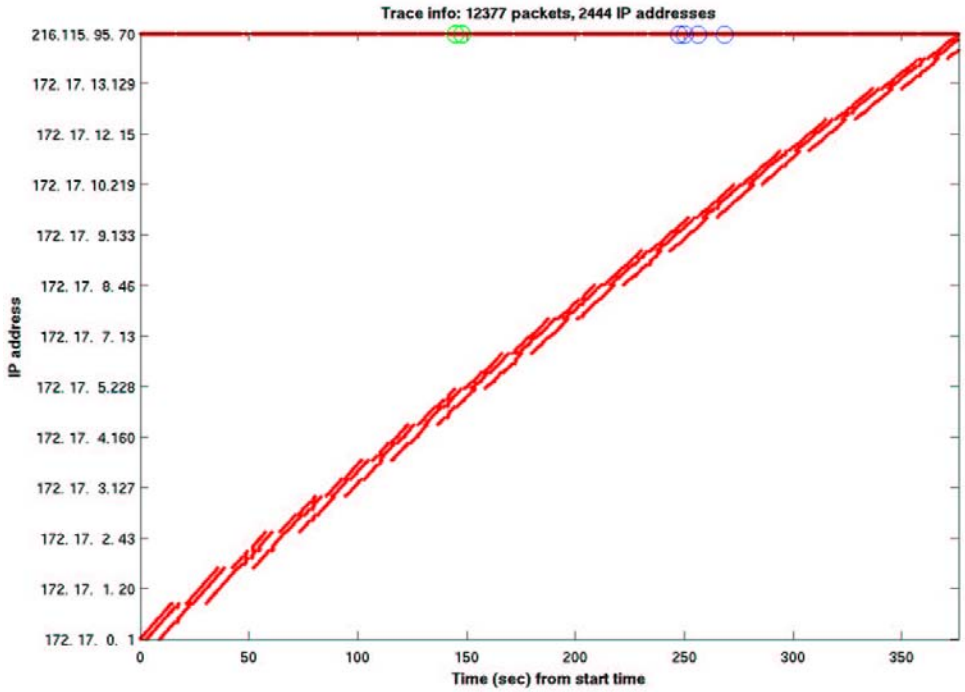

Figure 8: *Needle in a haystack. A large-scale scan attempts to conceal directed probes, shown here by the green and blue open circles.*

4.1.2.5 Coordinated Scans

Filename: scan_H_TCP_fast_seq_coordinated.tcp

Source: DREnet, 8 December 2002
Duration: 1h 1m 28.4s
Description: Three class B networks are scanned by 3 different hosts. A TCP SYN scan is performed simultaneously by three hosts, each scanning a single class B network for the same port. All scans occur approximately in sequence (see Figure 9). There are 167363 packets in the trace, 18 of which are responses to the scan (ICMP and TCP reset). The scan packets arrive at an average rate of 45.37 packets/second to 167327 unique IP addresses.

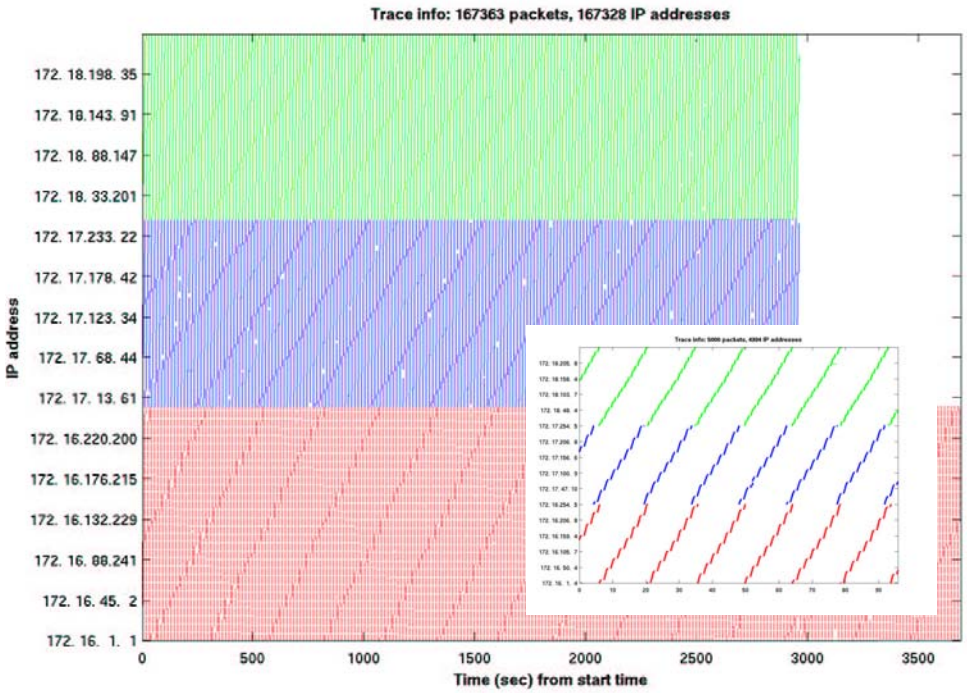

Figure 9: *Coordinated scan of 3 class B networks. The different origins are indicated by different colours.*

Filename: scan_V_TCP_fast_ran_coordinated.tcp
Source: Crafted (simDistPortScan1.pl)
Duration: 1.0s
Description: Each of 13 source hosts probes for a different port on one destination host. The probes occur over a 1 second interval with delay between packets at an average of 1/13s with 75% jitter, or within the interval [0.01925s, 0.13462s].

Filename: scan_S_TCP_fast_ran_coordinated.tcp
Source: Crafted (simDistPortScan2.pl)
Duration: 1.0s

Description: A group of 13 attackers coordinate their efforts in this attack. Each attacker chooses a single port to focus on, and scans the set of target hosts for this port. The time delay between probes (against 42 hosts) averages 1/545s with 75% jitter, or uniformly distributed in the interval [0.000459s, 0.003211s].

Filename: scan_H_TCP_fast_ran_coordinated.tcp
Source: Crafted (simDistPortScan3.pl)
Duration: 2.3s
Description: Each of 13 attackers target a different host on the target network. A single probe (TCP SYN to port 21) is sent from each attacker to its chosen target. The probes are spaced on average 2/13s apart with 75% jitter, or uniformly distributed on the interval [0.03846s, 0.26924s].

4.1.3 Traceroute

Filename: traceroute.tcp
Source: DREnet, 30 August 2000
Duration: 1m 25.6s
Description: UDP packets are sent from one host to one host, with the TTL starting at 1 and increasing until it reaches 17. This is consistent with the *traceroute* tool, used to determine the path that packets take to reach a host.

4.1.4 OS Identification

Filename: fingerprint.tcp
Source: DREnet, 30 August 2000
Duration: 0.1s
Description: One host is probed with packets consistent with the QueSO OS fingerprint tool: TCP packets with the SYN, SYN-ACK, FIN, FIN-ACK, SYN-FIN, PSH and finally SYN with the two reserved bits set. The responses generated include 2 RST packets.

4.1.5 Vulnerability Assessment Tools

Filename: cybercop.tcp
Source: DREnet, 30 August 2000
Duration: 27m 8.9s
Description: Pattern of behaviour corresponding to the CyberCop vulnerability assessment tool. It was originally designed to be used by a system administrator to lock down hosts on a network, but it may be used by an attacker to try to determine vulnerabilities.

4.1.6 Possible False Positives

The true purpose of the following traces cannot be positively determined. They may be TCP reset scans, or they may be backscatter effects as the result of either needle-in-the-haystack scanning activity or denial of service.

Filename: backscatter1.tcp
Source: DREnet, 29-30 August 2000
Duration: 23h 34m 38.3s
Description: Six unique IP addresses send 5330 TCP RST packets to 166 unique destination IP addresses on 3 class B networks. The source ports are randomly distributed between 900 and 9000, and the destination ports appear to be uniformly distributed between 0 and 65536. The packets arrive at an average rate of one packet every 15.9 seconds.

Filename: backscatter2.tcp
Source: DREnet, 29 April 2001
Duration: 21m 57.8s
Description: One source IP address sends 9 TCP RST packets to 9 unique destination IP addresses on 3 class B networks at an average rate of one packet every 15.9 seconds. The destination ports are >1023, as is all but one source port. After a 90 second pause, this activity is followed by 520 TCP RST packets sent to 520 unique IP address on 3 class B networks, with source port 0 and destination port either 1024 or 3072 at an average rate of one packet every 2.1 seconds.

4.2 Escalation

Password cracking and session hijacking were not found in the DREnet traffic and traces were incomplete in the DEFCON traffic. These types of attacks were simulated on a lab network or crafted.

4.2.1 Password Cracking

Filename: offline_passwd.tcp
Source: Closed network
Duration: 5h 39m 47.1s
Description: Offline password cracking. On a closed network, `/etc/passwd` and `/etc/shadow` are retreived via a misconfigured FTP server, the password for root is cracked locally, and then the attacker logs in as root via SSH at a later time. Note that the retrieval and SSH sessions are contained in 2 separate files in the merged versions due to the large interval of inactivity: offline_passwd_1.tcp and offline_passwd_2.tcp.

Filename: online_passwd.tcp

Source: Closed network
Duration: 5.2s
Description: Online password cracking. On a closed network, a scripted attack was run that reads each line of a password "dictionary" and sends this password as an argument to a mysql login attempt. Further description is contained in the script created to simulate the attack, `dictAttack.pl` (Section B.1).

4.2.2 Buffer Overflow

Filename: buffer.tcp
Source: DEFCON
Duration: 0.03s
Description: A buffer overflow is attempted on TCP port 53 in an attempt to get a shell with root access. In the content, it can be seen that the attacker has actually broken up the `/bin/sh` string to avoid detection by an IDS. However, the large number of x86 NOP instructions (hexadecimal value `0x90`) is a signature. Note that the merged data contains only a 144 byte data capture length.

4.2.3 Priveleged File Access

Filename: webprobe.tcp
Source: DREnet, 30 August 2000
Duration: 1h 34m 30.9s
Description: Attempts are made to access several dangerous files on a web server, such as `/cgi-bin/sh` and `scripts/cmd.exe`.

4.3 Denial of Service

Denial of service attacks were not found in the DREnet data and were incomplete in the DEFCON data. The reference data contains data collected on a closed network, generated using publicly-available tools.

4.3.1 Vulnerability DoS

The following traces contain denial of service attempts that rely on a vulnerability within the TCP/IP stack.

4.3.1.1 Land Attack DoS

Filename: land.tcp
Source: Closed network
Duration: N/A (single packet)
Description: The `land.c` code was compiled and launched on a closed network. The attack consists of sending a TCP SYN packet to the victim, where the return IP

address has been forged to be identical to the target IP address. The land attack will crash some older operating systems.

4.3.1.2 Teardrop DoS

Filename: teardrop.tcp
Source: Closed network
Duration: 0.9s
Description: The `teardrop.c` code was compiled and launched on a closed network. The attack involves IP fragmentation: an IP packet fragment is sent with an offset specified to overwrite the data supplied by the previous fragment. When the packet is reassembled at the target address, some older operating systems crash. The attack in this example uses two fragments of one packet, and the packet is sent repeatedly.

4.3.2 Multipacket DoS

The following traces contain denial of service attempts that rely on reseource starvation or on congestion.

4.3.2.1 Smurf DoS Attack

Several scripts and programs were written to craft traces which would have been generated by the Smurf tool (see Section B.3). The following attacks were simulated.

Filename: smurf1.tcp
Source: Crafted (simSmurf1.cc)
Duration: N/A (single packet)
Description: An unsuccessful attempt to use our network to smurf a target. An ICMP echo request packet is directed to a broadcast IP address, with no reply.

Filename: smurf2.tcp
Source: Crafted (simSmurf2a.cc simSmurf2b.cc simSmurf2.pl)
Duration: 16m 42.2s
Description: Our network is used to smurf a target. ICMP echo request packets are directed to a broadcast address at a rate of 10 per second, with delay between packets in the interval [0.07s, 0.13s] and a packet drop rate of 1%. Existing hosts reply to the spoofed source address. It is assumed that 25% of the available address space is used (60 hosts). Each host has its own characteristic delay d, ranging from 0.14ms to 0.5ms. Approximatly 10000 echo reqests are received. The hosts on the local network respond to each packet with a 0.5% chance of dropping the reply packet, with a delay between replies uniformly distributed in the interval $[d, 2d]$.

Filename: smurf3.tcp
Source: Crafted (simSmurf3a.pl)
Duration: 5m 1.7s
Description: A smurf attack is directed at us. 184059 ICMP echo reply packets are sent to us from a class C subnet containing 62 nodes. Each node sends packets with delay uniformly distributed within the interal [0.08s, 0.12s] (averaging 610 packets per second), with a packet drop rate of 1%.

4.3.2.2 TCP SYN Flood DoS

Filename: synful.tcp
Source: Closed network
Duration: 1.5s
Description: The exploit program `synful.c` was compiled and launched on a closed network. 20010 TCP SYN packets were sent to port 80 on one destination IP address at a rate of 13485.9 packets per second. The packets were spoofed to have 20010 unique source IP addresses; the random distribution of spoofed source IP addresses is shown in Figure 10.

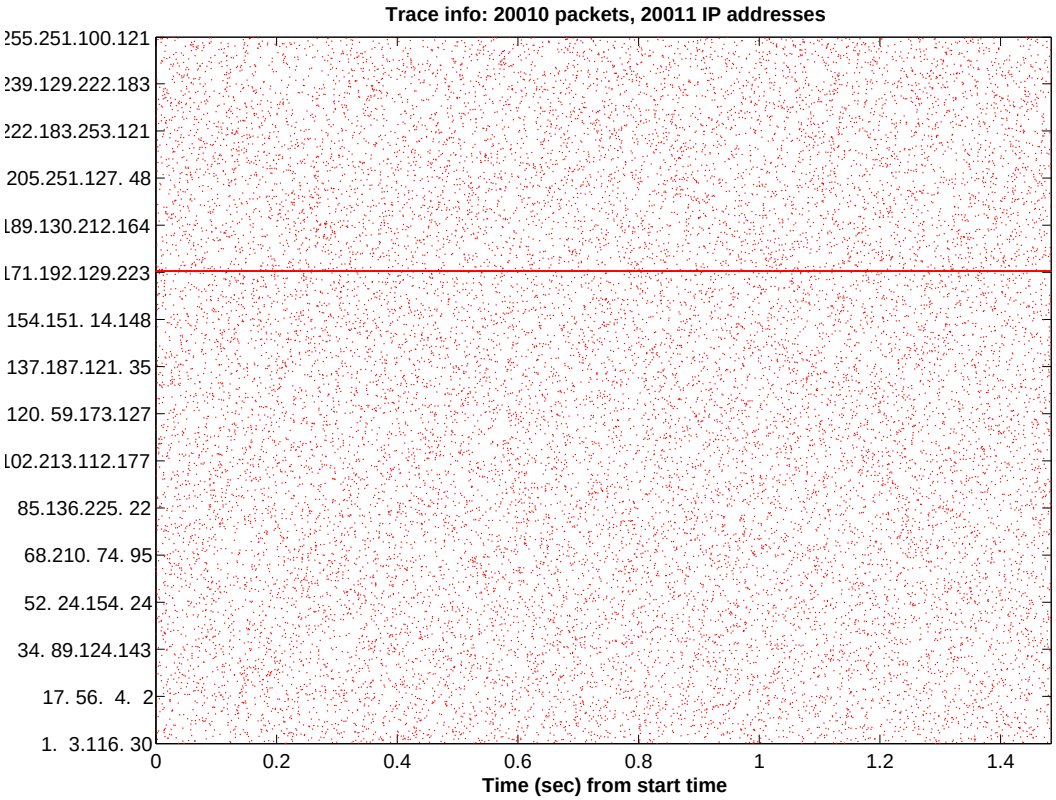

Figure 10: *A DoS performed by the tool* `synful`*. The uniform distribution of randomly generated source IP addresses are shown; the target is the solid line.*

4.3.2.3 Distributed Denial of Service

Filename: tfn2k.tcp
Source: Closed network and crafted (`even.pl`)
Duration: 1m 46.0s
Description: The DDoS exploit program *tribal flood network 2000* was compiled and launched on a closed network with one host acting as both the master and slave, and the other acting as victim. 250000 packets were collected five times, resulting in 5 separate traces representing 5 slaves. The TTL values for each trace were modified to reflect slaves positioned at 5, 8, 12, 17 and 20 hops distance from the victim. The files were merged and the timestamps stretched such that a 5 Mbps line would be just exceeded. This data was combined with the clean data, and packets were dropped to reflect the 5 Mbps limit. This data is presented as the merged data, and the DDoS traffic was extracted to give the clean DDoS data.

4.4 Covert Data Transfer

We include covert communications that can either send data or embed data within an image.

4.4.1 Covert Tunnels

The commands transferred via the tunnels in the traces are as follows:

1. echo “hello”
2. ls
3. pwd
4. ls /etc
5. cat /etc/passwd

4.4.1.1 ICMP Tunnels

The *loki* software had to be modified before it could be compiled: in `loki.h`, the ip.h and icmp.h include statements were reversed, and signals.h was changed to be from the `sys` directory rather than `linux`. The Makefile was modified to remove the DEBUG mode.

Filename: loki-nocrypt.tcp
Source: Closed network
Duration: 13.3s
Description: The *loki* ICMP tunnel software was compiled on a client and a server with no encryption. The daemon was started on the server, then the client executed remote commands.

Filename: loki-weakcrypt.tcp
Source: Closed network
Duration: 10.6s
Description: The *loki* ICMP tunnel software was compiled on a client and a server with weak encryption. The daemon was started on the server, then the client executed remote commands.

4.4.2 Information Hiding

The information hiding examples embed the secret message "The golden eagle has landed."

4.4.2.1 Steganography

The steganography tool *gifshuffle* [32] was used to embed a message in a gif image. The original image, `drdc-splash4.gif`, was modified to include the secret message. The resulting image is stored in `drdc-splash4-stego.gif`.

4.4.2.2 TCP/IP Fields

The `covert_tcp.c` code [20] was installed on a client and a server on a closed network.

Filename: covert_tcp_ipid_closedport.tcp
Source: Closed network
Duration: 28.3s
Description: The secret phrase was transferred using the IP ID header field. The data was sent to a closed port.

Filename: covert_tcp_ipid_openport.tcp
Source: Closed network
Duration: 2m 6.5s
Description: The secret phrase was transferred using the IP ID header field. The data was sent to an open port, resulting in a much longer trace due to the replays of SYN/ACK packets.

Filename: covert_tcp_seq_closedport_slower.tcp
Source: Closed network
Duration: 4m 40.3s
Description: The secret phrase was transferred using the TCP sequence number header field. The data was sent to a closed port. The source code was modified to increase the sleep time between packet transmissions.

Filename: covert_tcp_seq_openport.tcp

Source: Closed network
Duration: 2m 5.3s
Description: The secret phrase was transferred using the TCP sequence number header field. The data was sent to an open port, resulting in a much longer trace due to the replays of SYN/ACK packets.

Filename: covert_tcp_seq_bounced.tcp
Source: Closed network
Duration: 3m 19.5s
Description: The secret phrase was transferred using the TCP sequence number header field. The "bounced" option was used for this trace: the data was sent to an open port on an intermediate host, with forged source IP address. The replies were then directed from the intermediate host to the server for interpretation. The trace appears to originate from internal addresses.

5 Discussion

Collecting an exhaustive set of network attacks is not practical. In this work, we have attempted to collect a wide assortment of attacks, particularly attempting to cover a variety of attributes for the reconnaissance family. As the need arises, more attacks will be added to the data set, such as HTTP tunnels or man-in-the-middle PKI attacks. Wireless protocols could also be captured. This data is meant to serve as a starting point, to provide a basic set of known attacks to test IDS and visualisation methodologies.

By using the same set of ambient traffic to create the merged data set, we have the advantage of a controlled variable among the attacks. The drawback, of course, is that there is less variety in the types of legitimate ambient traffic. The cleaned ambient traffic may also have been over-pruned: the TCP FSM also removes traffic that is "normal" but not malicious, *e.g.* misconfigurations. This may not meet the user requirements. Even so, the ambient traffic cannot be guaranteed to be totally clean.

In removing the content of the ambient traffic, we also removed information that may be useful in testing the methods. This could not be helped as it was necessary to protect the privacy of DREnet users.

6 Conclusion and Recommendations

This report has documented the network attack reference data set created via custom software at DRDC. The software is capable of manipulating IP traffic traces in pcap format, including header fields such as IP address and time-to-live, as well as a variety of manipulations of the timestamp such as jitter, time shifting, compression or expansion. The software is also capable of merging two or more traffic traces with interleaved timestamps, essential for inserting attack traces into normal traffic.

The resulting network attack reference data set can be used to test the limits of intrusion detection systems and to test new intrusion detection techniques and network traffic visualisation techniques. The reference data set should be continually updated with new attacks. The capabilities of the software have been included in DRDC's Network Traffic Analysis Toolbox [33] as powerful packet and trace manipulation functions.

References

1. Bace, R. and Mell, P. (2001). Intrusion Detection Systems. *NIST Special Publications*, pp. 19–20.

2. Das, K. (2002). Protocol Anomaly Detection for Network-based Intrusion Detection (Online). SANS Institute. `http://www.sans.org/rr/intrusion/anomaly.php` (Jan. 2003).

3. Zissman, M. (2001). DARPA Intrusion Detection Evaluation (Online). MIT Lincoln Laboratory. `http://www.ll.mit.edu/IST/ideval/` (13 Apr 2004).

4. Dain, O.M. and Cunningham, R.K. (2001). Building Scenarios from a Heterogeneous Alert Stream. In *IEEE Workshop on Information Assurance and Security*, West Point, NY.

5. Staniford, S., Hoagland, J.A., and McAlerney, J.M. (2002). Practical Automated Detection of Stealthy Portscans. *Journal of Computer Security*, **10**, 105–136.

6. Yegneswaran, V., Barford, P., and Ullrich, J. (2003). Internet Intrusions: Global Characteristics and Prevalence. In *ACM SIGMETRICS'03*, San Diego, CA.

7. Fyodor (1997). The Art of Portscanning. *Phrack Magazine*, **7**(51). http://www.phrack.com.

8. Stevens, W. R. (1994). TCP/IP Illustrated : The Protocols, Vol. 1. Addison-Wesley.

9. Northcutt, S. (1999). Network Intrusion Detection: An Analyst's Handbook, Indianapolis, IN: New Riders.

10. Jordi Murgo (savage@apolstols.org) (Dec. 2000). Index of /pub/tools/unix/scanners/queso (Online). CERIAS Security Archive. `http://ftp.cerias.purdue.edu/pub/tools/unix/scanners/queso` (Sep 2003).

11. Fyodor (Feb. 2003). NMAP (Online). Insecure.org. `http://www.insecure.org/nmap` (Apr 2003).

12. Network Associates (2003). Online Vulnerability Assessment Service (Online). McAfee ASaP. `http://www.mcafeeasap.com/intl/EN/content/cybercop_asap/default.asp` (2 Feb 2004).

13. Naval Surface Warfare Center, Dahlgren Division. SHADOW Indications Technical Analysis: Coordinated Attacks and Probes (Online). http://www.nswc.navy.mil/ISSEC/CID/co-ordinated_analysis.txt (24 Sep 2003).

14. Solar Designer (Sept. 2003). John the Ripper password cracker (Online). http://www.openwall.com/john (22 Jan 2004).

15. HooBie Inc. (July 2002). Brutus - The Remote Password Cracker (Online). http://www.hoobie.net/brutus (January 22, 2004).

16. Baker, F. (1995). RFC 1812: Requirements for IP Version 4 Routers. (Online).

17. Homelien, O. (Jan. 2004). Smurf Amplifier Registry (Online). PowerTech Information Systems AS. http://www.powertech.no/smurf (29 Jan 2003).

18. Deitrich, S., Long, N., and Dittrich, D. (2000). Analyzing DDOS Tools: The Shaft Case. In *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, New Orleans, LA.

19. daemon9 (1997). LOKI2 (the implementation). *Phrack Magazine*, **7**(51). http://www.phrack.com.

20. Rowland, C.H.. Covert Channels in the TCP/IP Protocol Suite (Online). First Monday. http://www.firstmonday.dk/issues/issue2_5/rowland/ (25 Sep 2003).

21. Dark Tangent and BlackBeetle (Dec. 2003). Welcome to DEFCON (Online). http://www.defcon.org (2 Feb 2004).

22. The Tcpdump Group (Jan. 2002). Tcpdump (Online). http://www.tcpdump.org (28 Aug 2003).

23. (Apr. 2004). Ethereal: A Network Protocol Analyzer (Online). http://www.ethereal.com (13 Apr 2004).

24. Savage, S. (1999). Sting: a TCP-based Network Measurement Tool. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems*, pp. 71–79. Boulder, CO.

25. Snort (Online). http://www.snort.org (28 Aug 2003).

26. TCPSlice (Online). http://sourceforge.net/projects/tcpslice (28 Aug 2003).

27. Blanton, E. (Sept. 2002). TCPurify - a ”sanitary” sniffer (Online). http://masaka.cs.ohiou.edu/~eblanton/tcpurify (8 Mar 2004).

28. Whitehat (Online). http://www.whitehats.com/ids/vision18.rules.gz (28 Aug 2003).

29. (July 2000). Index of mirrors/shmoo/cctf-defcon8 (Online). wi2600.org. http://mediawhore.wi2600.org/mirrors/shmoo/cctf-defcon8 (28 Aug 2003).

30. Treurniet, J. and Lefebvre, J.H. (2003). A Finite State Machine Model of TCP Connections in the Transport Layer. (DRDC Ottawa TM 2003-139). Defence R&D Canada – Ottawa.

31. Whitehats, Inc. (2001). Whitehats Intrusion Detection Events Database: Full details for probe-myscan (Online). http://www.whitehats.com/info/IDS439 (10 Feb 2004).

32. Kwan, M. (Jan. 2003). The Gifshuffle Home Page (Online). Darkside Technologies Pty Ltd. http://www.darkside.com.au/gifshuffle (29 Jan 2004).

33. Gregoire, M. and Lefebvre, J. H. (in preparation). The Network Traffic Analysis Toolbox. Technical Report. DRDC Ottawa.

Annex A
TCPUtils Source Code

TCPUtils is a set of programs which were written to facilitate the manipulation of pcap files. They rely on TCPLib to read and write the packet traces.

A.1 TCPMerge (`tcpmerge.cc`)

TCPMerge is used to join several pcap files together. Unlike *Tcpslice*, the input files are allowed to have interleaved timestamps. Care should be taken to ensure that input files have the same packet capture length. *TCPTruncate* can be used to accomplish this.

A.2 TCPIPTranslate (`tcpiptranslate.cc`)

TCPIPTranslate is used to replace one IP in a trace with another. It will also recalculate the checksum to ensure proper packets are generated. More than one IP can be exchanged if there are more than one pair of IP addresses listed on the command line.

A.3 TCPTTLTranslate (`tcpttltranslate.cc`)

TCPTTLTranslate is used to alter the TTL (time to live) fields in IP packets. A value specified on the command line is added to the TTL fields in all IP packets of the specified trace. The IP checksum is recalculated to ensure a proper trace is produced. The value by which the TTL is modified may be negative.

A.4 TCPTimeShift (`tcptimeshift.cc`)

TCPTimeShift modifies time stamps in a packet trace to make the trace appear as if it occurred at a different time. All time stamps are altered by the same amount (specified on the command line). Time stamps may be moved either forward or backward in time.

A.5 TCPTimeStretch (`tcptimestretch.cc`)

TCPTimeStretch will scale the difference in time between packets. This can be used to compress or expand traces to simulate the appearance of a faster or slower connection. The time stamp of the first packet is preserved.

A.6 TCPTruncate (`tcptrunc.cc`)

TCPTruncate will reduce the captured data length in pcap files. This is useful if the merging of files is desired, and one trace has a higher capture length (pcap snap length) than the other.

A.7 TCPJitter (`tcpjitter.cc`)

TCPJitter can be used to add randomness to packet traces. This is especially useful when working with crafted traces, as the results of such crafting are usually idealized, and randomness can make them appear more realistic. This program can accept two types of arguments — a time jitter factor, and a drop rate. If a time jitter factor is specified on the command line, the program will randomly add or subtract time between packets, up to a maximum specified percentage. If a drop rate is specified, then each packet will have this percentage chance of being dropped (left out of the output trace).

A.8 TCPRate (`tcprate.cc`)

Useful for analyzing potential packet flood attacks, *TCPRate* will output a table of bytes transfered in each second of a trace. This table is suitable for plotting in programs like *gnuplot*.

A.9 TCPContent (`tcpcontent.cc`)

TCPContent will output the ASCII representation of TCP or IP packets. This utility is extremely useful for determining an attacker's actions when a text based, non-encrypted protocol (like HTTP or FTP) is used.

A.10 TCPTimeSpace (`tcptimespace.cc`)

The *TCPTimeSpace* utility will evenly space out packets occurring in the same second. This is particularly useful when working with files generated by scripts which have many different packets occurring at exactly the same point in time (such as the coordinated scans). *TCPTimeSpace* is typically used with *TCPJitter* to simulate packets arriving at various points in time.

Annex B
Simulation Script and Program Source Code

Many of the simulated attacks were performed using Perl scripts and custom C++ programs. Also listed in this section is the `Makefile` which automates the build process of the TCPLib, TCPUtils and C++ attack simulation utilities.

B.1 Online Password Cracking

B.1.1 dictAttack.pl

```
#!/usr/bin/perl

# Jason McKenna, summer 2003.

# Script to help simulate a dictionary attack
# Will read in passwords from file, pass them to mysql client which will attempt
# to login using the password.  A return code of 0 means the password was
# correct.  We assume that the attacker already knows:
# a) there is a mysql server running on the port
# b) there is a database named "data"
# c) there is a user named "jason"
# d) jason has permission to login from anywhere from the Internet

# open the password file
open FH, "passwords";

# set flag to false (not really required as already 0 but...)
#  and it's better practice to exit at end of program, rather then quit in
#  middle
$done = 0;

# read lines while we haven't found password
while (($pass = <FH>) && ($done == 0)) {

#strip endline from password
$pass =~ s/\n//;

#try to log in (and quit if successful)
system "echo \\\\q | mysql -u jason -p$pass -h 192.168.2.8 data > /dev/null
2>&1";
if ($? == 0) {
print "Found password: $pass\n";
$done = 1;
}
}

if ($done == 0) {
print "Could not find password.\n";
}
```

B.2 Scan Generators

B.2.1 simDistPortScan1.pl

```
#!/usr/bin/perl

#Simulate many hosts scanning 1 host, many ports
```

```
#amount to jitter the timestamps
$jitter = 75;

#modified to obsificate DREnet IP
$net = "172.168";

# these addresses are random and may or may not resolve to actual computers
# on the internet
%scanners = ("24.42.204.111" => 7, "24.56.221.241" => 21, "181.61.24.66" =>23, "198.224.55.10"=>53, "17.22.120.134" => 8

$ver = 1;

#what hosts on $net to simulate the attack against.
@targets = ("251.1");

$outfile = "distPortScan$ver.tcp";

#each attacer will probe the appropiate ports on each victim
foreach $attacker (keys %scanners) {
foreach $victim (@targets) {
$file = "distPortScan$ver.$attacker.$victim.tcp";
print "$attacker scanning $net.$victim port @scanners{$attacker}\n";
system ("./simPortScan $file $attacker $net.$victim @scanners{$attacker} @scanners{$attacker} 1");
$filelist .= " $file";
}
}

#create thie final file by merging together all the generated files
system "tcpmerge $filelist $outfile";
system "rm $filelist";
system "tcptimespace $outfile a$outfile"; #space out the packets evenly (impose gaps between packets occurint at same ti
system "tcpjitter a$outfile $outfile t$jitter";
```

B.2.2 simDistPortScan2.pl

```
#!/usr/bin/perl

#Jason McKenna, summer 2003

#Simulate many hosts scanning many hosts, many ports
$jitter = 75;

#modified to obsificate DREnet IP
$net = "172.168";

%scanners = ("24.42.204.111" => 7, "24.56.221.241" => 21, "181.61.24.66" =>23, "198.224.55.10"=>53, "17.22.120.134" => 8
$ver = 2;
@targets = ("251.1" ,"240.145","240.150","240.151","242.4","242.3","242.18","242.200",,"241.16","241.4","244.2","244.3",

$outfile = "distPortScan$ver.tcp";

#generate the scan that each attacker performs against each victum
foreach $attacker (keys %scanners) {
foreach $victim (@targets) {
$file = "distPortScan$ver.$attacker.$victim.tcp";
print "$attacker scanning $net.$victim port @scanners{$attacker}\n";
system ("./simPortScan $file $attacker $net.$victim @scanners{$attacker} @scanners{$attacker} 1");
$filelist .= " $file";
}
}

system "tcpmerge $filelist $outfile";
```

```
system "rm $filelist";
system "tcptimespace $outfile a$outfile";
system "tcpjitter a$outfile $outfile t$jitter";
```

B.2.3 simDistPortScan3.pl

```
#!/usr/bin/perl

# Jason McKenna, summer 2003
$jitter = 75;

#modified to obsificate DREnet IP
$net = "172.168";

#Simulate many hosts scanning many hosts, 1 port
%scanners = ("24.42.204.111" => "240.151", "24.56.221.241" => "242.4", "181.61.24.66" =>"242.3", "198.224.55.10"=>"242.1
$ver = 3;
$port = 21;

$outfile = "distPortScan$ver.tcp";

#each attacker scans for port 21 on the host it is associated with in the hash
foreach $attacker (keys %scanners) {
$victim = @scanners{$attacker};
$file = "distPortScan$ver.$attacker.$victim.tcp";
print "$attacker scanning $net.$victim port $port\n";
system ("./simPortScan $file $attacker $net.$victim $port $port 1");
$filelist .= " $file";
}

#merge all files
system "tcpmerge $filelist $outfile";
system "rm $filelist";
system "tcptimespace $outfile a$outfile";
system "tcpjitter a$outfile $outfile t$jitter";
```

B.2.4 simPortScan.cc

```
/* Copyright
* (C) Her Majesty the Queen, as represented by the Minister of National Defence,
* 2003
*
* (C) Sa majeste la reine, representee par le ministre de la Defense nationale,
* 2003
*
* Written by Jason McKenna, summer 2003 for DRDC
*/

/* simPortScan.cc
Simulates a port scan from one host vs another.

Specified IP will scan another specified IP for open TCP ports in a range
also specified on the command line.

usage:

simPortScan <file> <sourceIP> <destIP> <startPort> <endPort> <speed> [-r]

where <file> is the file created, <sourceIP> and <destIP> are the IPs involved,
<startPort> and <endPort> specify the rang eof ports to scan, <speed> determines
the rate at which to scan, and the optional [-r] randomised the order in which
```

```
to scan.

*/

#include "tcplib.h" //use the TCPLib to write PCap files.
#include <iostream.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

#define TTL 111 //default TTL if none specified on cmd line
#define PACKETLEN 54
#define TIME_STAMP_START 1055936617 // About 7:45, June 18, 2003
#define RAND_DELTA 0x7fffffff //used to flag that you want random generated
#define IP_ID_INIT 25338
#define IP_ID_DELTA 256 //256 is typical for Win machines, 1 or random typical
//for *nix -- set to RAND_DELTA if you want a random ip id delta
#define IP_TOS 0x10
#define IP_DONT_FRAG true
#define IP_MORE_FRAG false
#define TCP_SRC_PORT_INIT 3605
#define TCP_SRC_PORT_DELTA RAND_DELTA //set to RAND_DELTA for random src port
#define TCP_SYN true
#define TCP_SEQ_INIT 48837445
#define TCP_SEQ_DELTA 15 //set to RAND_DELTA for random seq #
#define TCP_WINDOW 8192
#define POS_FILE 1
#define POS_SOURCE 2
#define POS_DEST 3
#define POS_START 4
#define POS_END 5
#define POS_SPEED 6
#define POS_RANDOM 7

void displayHelp();

int main(int argc, char ** argv) {
struct timeval tm;
gettimeofday(&tm,NULL);
srand(getpid());
srand(tm.tv_usec);
uint8_t smac[6] = {0x00, 0x08, 0xe3, 0x17, 0xd0, 0x90 }; //source mac
// for packets (hardware addr of router)
uint8_t dmac[6] = {0x00, 0xe0, 0x1e, 0xa5, 0x14, 0xe2 }; //dest mac for
// packets (harware addr of firewall or server, depending
// on how firewall is configured)

//did user ask for help
for (int i = 1; i < argc; i++) {
if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i],"--help") == 0) {
cout << "a" << endl;
displayHelp();
return 0;
}
}

//did user enter incorrect args
if ((argc != 7)  && (argc != 8)) {
cout << "b" << endl;
displayHelp();
return 1;
}
```

```
//get the first and last port
int startPort = atoi(argv[POS_START]);
int endPort = atoi(argv[POS_END]);
if (endPort < startPort) {
int temp = endPort;
endPort = startPort;
startPort = temp;
}

//fill a "port" array with port numbers
uint16_t port[endPort - startPort + 1];
for (int i = startPort; i <= endPort; i++) {
port[i-startPort] = i;
}

//if the user specifed an extra arg on cmd line...
if (argc == 8) {
//if arg was "-r", randomize order of ports
if (strcmp(argv[POS_RANDOM], "-r") == 0) {
for (int i = 0; i <= (endPort - startPort); i++) {
int swapPos = rand() % (endPort - startPort);
uint16_t temp = port[i];
port[i] = port[swapPos];
port[swapPos] = temp;
}
//otherwise, display help
} else {
displayHelp();
return 1;
}
}

//read IPs from cmd line
uint32_t sourceIP = strToIP(argv[POS_SOURCE]);
uint32_t destIP = strToIP(argv[POS_DEST]);

//read the speed from the command line
float fspeed = atof(argv[POS_SPEED]);
if (fspeed == 0.0) {
//if user entered invalid speed (0 or non-numeric);
displayHelp();
return 1;
}
if (fspeed < 0) fspeed = -fspeed;

int delays = (int) (1.0/fspeed);
int delayu = (int) (1000000.0 / (fspeed - ((float) delays)));

//seed the randomizer
srand(time(NULL));

//create the file header to be used in the output file
struct pcap_file_header fh;
fh.magic = TCPDUMP_MAGIC;
fh.pcap_version_major=PCAP_VERSION_MAJOR;
fh.pcap_version_minor=PCAP_VERSION_MINOR;
fh.thiszone = 0;
fh.sigfigs = 0;
fh.snaplen = 68;
fh.linktype = 1; //ethernet


TcplibFileWriter * writer = new TcplibFileWriter(argv[POS_FILE], fh);
```

```
TcplibTCPPacket * p = new TcplibTCPPacket(PACKETLEN);
struct timeval timestamp;
gettimeofday(&timestamp, NULL);
timestamp.tv_sec = TIME_STAMP_START;

//set ethernet header info (protocol set by ip constructor)
p->setEthernetSourceMAC(smac);
p->setEthernetDestMAC(dmac);

//ip ver, ihl, tos, tot_len, flags, frag_offset, protocol set by
// constructor
p->setIPSourceAddress(sourceIP);
p->setIPDestAddress(destIP);
p->setIPTTL(TTL);
p->setIPDontFragBit(IP_DONT_FRAG);
p->setIPMoreFragBit(IP_MORE_FRAG);
p->setIPTypeOfService(IP_TOS);
int id = IP_ID_INIT;
p->setTCPSynFlag(TCP_SYN);
p->setTCPWindow(TCP_WINDOW);
int sPort = TCP_SRC_PORT_INIT;
if (TCP_SRC_PORT_DELTA == RAND_DELTA) sPort = rand() % 65536;
int seq = TCP_SEQ_INIT;

//for each port in out list...
for (int i = 0; i <= (endPort -startPort); i++) {
p->setTimestamp(timestamp);
p->setTCPDestPort(port[i]);
p->setTCPSourcePort(sPort);
p->setTCPSeqNum(seq);
p->setIPID(id);
p->setIPChecksum(p->calculateIPChecksum());
int sum = p->calculateTCPChecksum();
p->setTCPChecksum(sum);
writer->writePacket(p);

//for debug purposes
//cout << "Wrote packet " << i << endl;

//get packet readey for next write;

//get next timestamp
timestamp.tv_sec += delays;
timestamp.tv_usec += delayu;
if (timestamp.tv_usec >= 1000000) {
timestamp.tv_usec -= 1000000;
timestamp.tv_sec++;
}

//select new source port
if (TCP_SRC_PORT_DELTA == RAND_DELTA) {
sPort = rand() % 65536;
} else {
//increment, and loop if appropiate
sPort += TCP_SRC_PORT_DELTA;
if (sPort >= 65536) {
sPort -= 65536;
}
}

//select new tcp seqence #
if (TCP_SEQ_DELTA == RAND_DELTA) {
seq = rand();
} else {
```

```
//increment, and loop if appropiate
seq += TCP_SEQ_DELTA;
}

//select new ip id
if (IP_ID_DELTA == RAND_DELTA) {
id = rand() % 65536;
} else {
//increment, and loop if appropiate
id += IP_ID_DELTA;
if (id >= 65536) {
id -= 65536;
}
}

}
delete p;
delete writer;
cout << "Successful." << endl;
return 0;
}

void displayHelp() {
cout << "simPortScan: Generates a port scan vs a host" << endl << endl

     << "usage: simPortScan <file> <sourceIP> <destIP> <startPort> <endPort> <speed> [-r]" << endl << endl

     << "  file: name of file to output." << endl
     << "  source/destIP: IP address of attacker/target" << endl
     << "  start/endPort: Port to start & end scan" << endl
     << "  speed: Scan speed in packets/second" << endl
     << "  -r: (optional) randomize scan order" << endl << endl

     << "Compiled with TCPLib " << TCPLIB_VERSION_MAJOR << "." << TCPLIB_VERSION_MINOR << "." << TCPLIB_VERSION_EXTRA <<
}
```

B.2.5 simHPortScan.cc

```
/* Copyright
* (C) Her Majesty the Queen, as represented by the Minister of National Defence,
* 2003
*
* (C) Sa majeste la reine, representee par le ministre de la Defense nationale,
* 2003
*
* Written by Jason McKenna, summer 2003 for DRDC
*/

/* simHPortScan.cc
Simulates a horizontal port scan from one host vs a range of hosts.

Specified IP will scan another specified IPs for open UDP port.

usage:

simHPortScan <file> <sourceIP> <startDestIP> <endDestIP> <port> <speed> [-r]

*/

#include "tcplib.h" //use the TCPLib to write PCap files.
#include <iostream.h>
#include <stdlib.h>
#include <sys/time.h>
```

```
#include <time.h>
#include <unistd.h>

#define TTL 111 //default TTL if none specified on cmd line
#define PACKETLEN 54
#define TIME_STAMP_START 1055936617 // About 7:45, June 18, 2003
#define RAND_DELTA 0x7fffffff //used to flag that you want random generated
#define IP_ID_INIT 25338
#define IP_ID_DELTA 256 //256 is typical for Win machines, 1 or random typical
//for *nix -- set to RAND_DELTA if you want a random ip id delta
#define IP_TOS 0x10
#define IP_DONT_FRAG true
#define IP_MORE_FRAG false
#define UDP_SRC_PORT_INIT 3605
#define UDP_SRC_PORT_DELTA RAND_DELTA //set to RAND_DELTA for random src port
#define POS_FILE 1
#define POS_SOURCE 2
#define POS_STARTIP 3
#define POS_ENDIP 4
#define POS_PORT 5
#define POS_SPEED 6
#define POS_RANDOM 7

void displayHelp();

int main(int argc, char ** argv) {
struct timeval tm;
gettimeofday(&tm,NULL);
srand(getpid());
srand(tm.tv_usec);
uint8_t smac[6] = {0x00, 0x08, 0xe3, 0x17, 0xd0, 0x90 }; //source mac
// for packets (hardware addr of router)
uint8_t dmac[6] = {0x00, 0xe0, 0x1e, 0xa5, 0x14, 0xe2 }; //dest mac for
// packets (harware addr of firewall or server, depending
// on how firewall is configured)

//did user ask for help
for (int i = 1; i < argc; i++) {
if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i],"--help") == 0) {
displayHelp();
return 0;
}
}

//did user enter incorrect args
if ((argc != 7)  && (argc != 8)) {
displayHelp();
return 1;
}

//read IPs from cmd line
uint32_t sourceIP = strToIP(argv[POS_SOURCE]);
uint32_t startDestIP = strToIP(argv[POS_STARTIP]);
uint32_t endDestIP = strToIP(argv[POS_ENDIP]);

if(ntohs(startDestIP) > ntohs(endDestIP)) {
uint32_t temp = startDestIP;
startDestIP = endDestIP;
endDestIP = temp;
}


//fill a "destIP" array with port numbers
uint32_t destIP[ntohl(endDestIP) - ntohl(startDestIP) + 1];
```

```
for (int i = ntohl(startDestIP); i <= ntohl(endDestIP); i++) {
destIP[i-ntohl(startDestIP)] = htonl(i);
}

//get the port
int port = atoi(argv[POS_PORT]);

//if the user specifed an extra arg on cmd line...
if (argc == 8) {
//if arg was "-r", randomize order of ports
if (strcmp(argv[POS_RANDOM], "-r") == 0) {
for (int i = 0; i <= ntohl(endDestIP) - ntohl(startDestIP); i++) {
int swapPos = rand() % (ntohl(endDestIP) - ntohl(startDestIP));
uint32_t temp = destIP[i];
destIP[i] = destIP[swapPos];
destIP[swapPos] = temp;
}
//otherwise, display help
} else {
displayHelp();
return 1;
}
}


//read the speed from the command line
float fspeed = atof(argv[POS_SPEED]);
if (fspeed == 0.0) {
//if user entered invalid speed (0 or non-numeric);
displayHelp();
return 1;
}
if (fspeed < 0) fspeed = -fspeed;

int delays = (int) (1.0/fspeed);
int delayu = (int) (1000000.0 / (fspeed - ((float) delays)));

//seed the randomizer
srand(time(NULL));

//create the file header to be used in the output file
struct pcap_file_header fh;
fh.magic = TCPDUMP_MAGIC;
fh.pcap_version_major=PCAP_VERSION_MAJOR;
fh.pcap_version_minor=PCAP_VERSION_MINOR;
fh.thiszone = 0;
fh.sigfigs = 0;
fh.snaplen = 68;
fh.linktype = 1; //ethernet


TcplibFileWriter * writer = new TcplibFileWriter(argv[POS_FILE], fh);

TcplibUDPPacket * p = new TcplibUDPPacket(PACKETLEN);
struct timeval timestamp;
gettimeofday(&timestamp, NULL);
timestamp.tv_sec = TIME_STAMP_START;

//set ethernet header info (protocol set by ip constructor)
p->setEthernetSourceMAC(smac);
p->setEthernetDestMAC(dmac);

//ip ver, ihl, tos, tot_len, flags, frag_offset, protocol set by
// constructor
```

```
p->setIPSourceAddress(sourceIP);
//p->setIPDestAddress(destIP);
p->setIPTTL(TTL);
p->setIPDontFragBit(IP_DONT_FRAG);
p->setIPMoreFragBit(IP_MORE_FRAG);
p->setIPTypeOfService(IP_TOS);
int id = IP_ID_INIT;
int sPort = UDP_SRC_PORT_INIT;
if (UDP_SRC_PORT_DELTA == RAND_DELTA) sPort = rand() % 65536;

//for each ip in out list...
for (int i = 0; i <= (ntohl(endDestIP) - ntohl(startDestIP)); i++) {
p->setTimestamp(timestamp);
p->setIPDestAddress(destIP[i]);
p->setUDPDestPort(port);
p->setUDPSourcePort(sPort);
p->setIPID(id);
p->setIPChecksum(p->calculateIPChecksum());
//int sum = p->calculateUDPChecksum();
p->setUDPChecksum(p->calculateUDPChecksum());
writer->writePacket(p);

//for debug purposes
//cout << "Wrote packet " << i << endl;

//get packet readey for next write;

//get next timestamp
timestamp.tv_sec += delays;
timestamp.tv_usec += delayu;
if (timestamp.tv_usec >= 1000000) {
timestamp.tv_usec -= 1000000;
timestamp.tv_sec++;
}

//select new source port
if (UDP_SRC_PORT_DELTA == RAND_DELTA) {
sPort = rand() % 65536;
} else {
//increment, and loop if appropiate
sPort += UDP_SRC_PORT_DELTA;
if (sPort >= 65536) {
sPort -= 65536;
}
}

//select new tcp seqence #
/*if (TCP_SEQ_DELTA == RAND_DELTA) {
seq = rand();
} else {
//increment, and loop if appropiate
seq += TCP_SEQ_DELTA;
}*/

//select new ip id
if (IP_ID_DELTA == RAND_DELTA) {
id = rand() % 65536;
} else {
//increment, and loop if appropiate
id += IP_ID_DELTA;
if (id >= 65536) {
id -= 65536;
}
}
```

```
}
delete p;
delete writer;
cout << "Successful." << endl;
return 0;
}

void displayHelp() {
cout << "simHPortScan: Generates a horizontal UDP scan vs a host range" << endl << endl

     << "usage: simPortScan <file> <sourceIP> <startDestIP> <endDestIP> <port> <speed> [-r]" << endl << endl

     << "  file: name of file to output." << endl
     << "  sourceIP/startDestIP/endDestIP: IP addresses of attacker/targets" << endl
     << "  port: Port to scan" << endl
     << "  speed: Scan speed in packets/second" << endl
     << "  -r: (optional) randomize scan order" << endl << endl

     << "Compiled with TCPLib " << TCPLIB_VERSION_MAJOR << "." << TCPLIB_VERSION_MINOR << "." << TCPLIB_VERSION_EXTRA <<
}
```

B.3 Smurf DoS Generators

B.3.1 simSmurf1.cc

```
/* Copyright
* (C) Her Majesty the Queen, as represented by the Minister of National Defence,
* 2003
*
* (C) Sa majeste la reine, representee par le ministre de la Defense nationale,
* 2003
*
* Written by Jason McKenna, summer 2003 for DRDC
*/

/* Used to simulate incomming packets to launch a Smurf attack vs another
* machine.
*
* usage: simSmurf1 <outfile>
*/

#include "tcplib.h"
#include <sys/time.h>
#include <stdlib.h>

//This is used to define the length of the text of the echo message
#define ECHO_MSG_LEN 190

//The IP was choosen randomly from Dec 2002 traffic
// 210.103.139.129 translates to (net byte order) 0xd2678b81
#define SOURCEIP 0xd2678b81
//This MAC was associated with the address above from the 2002 traffic.
// This is just the MAC of the next link in the chain connecting to the host
#define SOURCEMAC { 0x00, 0x08, 0xe3, 0x17, 0xd0, 0x90 }
// Modified to obsificate DREnet IPs
#define DESTIP 0xACA8ffff
// MAC of the router
#define DESTMAC { 0x00, 0x30, 0x80, 0xce, 0xba, 0xa2 }

int main(int argc, char ** argv) {
if (argc != 2) {
cout << "Error: you must specify one (1) output file." << endl;
```

```
return 1;
} else {
//Generate file header
}
//create pcap file header
pcap_file_header fh;
fh.magic = TCPDUMP_MAGIC; //tcpdump magic number
fh.pcap_version_major = 2; //major pcap version
fh.pcap_version_minor = 4; //minor pcap version
fh.thiszone = fh.sigfigs = 0;
fh.snaplen = 68; //default 68 byte capture length
fh.linktype = 1; //ethernet
//this is what we'll be using to write to the file
TcplibFileWriter * writer = new TcplibFileWriter(argv[1], fh);

//create the packet header
pcap_packet_header ph;

struct timeval timestamp;
gettimeofday(&timestamp, NULL);

ph.ts = timestamp;
ph.caplen = fh.snaplen; //caplen <= snaplen
ph.len = ETHER_HDR_LEN + sizeof(iphdr) + sizeof(icmphdr) + ECHO_MSG_LEN;

TcplibICMPPacket * p = new TcplibICMPPacket(fh.snaplen);
p->setRawHeader(ph);

//generate ethernet header
//generate random MACs
uint8_t smac[ETHER_ADDR_LEN] = SOURCEMAC;
uint8_t dmac[ETHER_ADDR_LEN] = DESTMAC;
//set MACs & protocol type
p->setEthernetSourceMAC(smac);
p->setEthernetDestMAC(dmac);

//set up the IP header
p->setIPVersion(IPVERSION);
p->setIPHeaderLength(5);
p->setIPSourceAddress(ntohl(SOURCEIP)); //make the apparent target of the smurf
p->setIPDestAddress(ntohl(DESTIP)); //send packet to broadcast address
p->setIPPacketLength(sizeof(iphdr) + sizeof(icmphdr) + ECHO_MSG_LEN);
p->setIPTTL(242); //TTL initially set to 255 in original smurf.c
p->setIPDontFragBit(true); //should be the only IP flag set
p->setIPChecksum(p->calculateIPChecksum());
p->setICMPType(ICMP_ECHO);

writer->writePacket(p);
delete p;
delete writer;
return 0;
}
```

B.3.2 simSmurf2a.cc

```
/* Used to simulate incomming packets to launch a Smurf attack vs another
* machine.
*
* usage: simSmurf2a <outfile> <numpackets> [-r] <ip> <trigger file> <delay>
*/
```

```
#include "tcplib.h"
#include <sys/time.h>
#include <stdlib.h>

//This is used to define the length of the text of the echo message
#define ECHO_MSG_LEN 190

//The IP was choosen randomly from Dec 2002 traffic
// 210.103.139.129 translates to (net byte order) 0xd2678b81
#define SOURCEIP 0xd2678b81
//This MAC was associated with the address above from the 2002 traffic.
// This is just the MAC of the next link in the chain connecting to the host
#define SOURCEMAC { 0x00, 0x08, 0xe3, 0x17, 0xd0, 0x90 }
// Modified to obsificate DREnet IPs
#define DESTIP 0xACA8ffff
// MAC of the router
#define DESTMAC { 0x00, 0x30, 0x80, 0xce, 0xba, 0xa2 }

#define POS_FILE 1
#define POS_NUM_PACKETS 2
#define POS_DELAY 3
#define POS_REPLY 4
#define POS_IP 5
#define POS_TRIGGER 6
#define POS_INIT_DELAY 7

void displayHelp();

int main(int argc, char ** argv) {
//check to see if user requested help
for (int i = 1; i < argc; i++) {
if ((strcmp(argv[i], "-h") == 0) ||(strcmp(argv[i], "--help") == 0)) {
displayHelp();
return 0;
}
}

if ((argc != 4) && (argc != 8)) {
displayHelp();
return 1;
} else {
uint32_t sourceip;
bool replyMode = false;
int numPackets = atoi(argv[POS_NUM_PACKETS]);
int delay = atoi(argv[POS_DELAY]);
if ((numPackets == 0) || (delay == 0)) {
displayHelp();
return 1;
} else {
struct timeval timestamp;
int initialDelay = 0;
if (argc == 8) {
replyMode = true;
initialDelay = atoi(argv[POS_INIT_DELAY]);
TcplibFileReader * trigger = new TcplibFileReader(argv[POS_TRIGGER]);
timestamp = trigger->getNextPacket()->getTimestamp();
timestamp.tv_usec += (initialDelay);
if (timestamp.tv_usec >= 1000000) {
timestamp.tv_sec++;
timestamp.tv_usec -= 1000000;
}

sourceip = strToIP(argv[POS_IP]);
delete trigger;
```

```
} else {
gettimeofday(&timestamp, NULL);
}

//create pcap file header
pcap_file_header fh;
fh.magic = TCPDUMP_MAGIC; //tcpdump magic number
fh.pcap_version_major = 2; //major pcap version
fh.pcap_version_minor = 4; //minor pcap version
fh.thiszone = fh.sigfigs = 0;
fh.snaplen = 68; //default 68 byte capture length
fh.linktype = 1; //ethernet
//this is what we'll be using to write to the file
TcplibFileWriter * writer = new TcplibFileWriter(argv[POS_FILE], fh);

//create the packet header
pcap_packet_header ph;


ph.ts = timestamp;
ph.caplen = fh.snaplen; //caplen <= snaplen
ph.len = ETHER_HDR_LEN + sizeof(iphdr) + sizeof(icmphdr) + ECHO_MSG_LEN;

TcplibICMPPacket * p = new TcplibICMPPacket(fh.snaplen);
p->setRawHeader(ph);

//generate ethernet header
uint8_t smac[ETHER_ADDR_LEN] = SOURCEMAC;
uint8_t dmac[ETHER_ADDR_LEN] = DESTMAC;


//set up the IP header
p->setIPVersion(IPVERSION);
p->setIPHeaderLength(5);
if (!replyMode) {
cout << "Entering non-reply mode." << endl;
p->setIPSourceAddress(ntohl(SOURCEIP)); //make the apparent target of the smurf
p->setIPDestAddress(ntohl(DESTIP)); //send packet to broadcast address
p->setIPTTL(242); //TTL initially set to 255 in original smurf.c
p->setICMPType(ICMP_ECHO);
p->setEthernetSourceMAC(smac);
p->setEthernetDestMAC(dmac);
} else {
cout << "Entering reply mode." << endl;
p->setIPSourceAddress(sourceip);
p->setIPDestAddress(ntohl(SOURCEIP));
p->setIPTTL(254); //TTL initially set to 255 in original smurf.c
p->setICMPType(ICMP_ECHOREPLY);
p->setEthernetSourceMAC(dmac);
p->setEthernetDestMAC(smac);
}
p->setIPPacketLength(sizeof(iphdr) + sizeof(icmphdr) + ECHO_MSG_LEN);
p->setIPDontFragBit(true); //should be the only IP flag set
p->setIPChecksum(p->calculateIPChecksum());
for (int i = 0; i < numPackets; i++) {
writer->writePacket(p);
timestamp.tv_usec += (delay);
if (timestamp.tv_usec >= 1000000) {
timestamp.tv_sec++;
timestamp.tv_usec -= 1000000;
}
p->setTimestamp(timestamp);
}
delete p;
```

```
delete writer;
return 0;
}
}
}

void displayHelp() {
cout << "simSmurf2a:" << endl << endl

     << "Generates a trace as if an attacker was sending several packets to us, in an" << endl
     << "  attempt to flood the target (spoofed source ip).  Our fw blocks ICMP." << endl << endl

     << "usage: simSmurf2a <outfile> <number of packets> <time between packets (us)>" << endl
     << "       [-r <ip> <trigger file> <delay>]" << endl << endl
     << "       where the presence of -r indicates to generate echo replys instead of" << endl
     << "       echo requests.  The <delay> argument specified the delay from the" << endl
     << "       original timestamp (the first packet in the <trigger file>) to generate" << endl
     << "       the reponses.  <ip> is the IP responding." << endl;
}
```

B.3.3 simSmurf2b.cc

```
/* Reply to smurf packets incomming.
*
* usage: simSmurf2b <trigger file> <outfile> <host> <min delay> <max delay>
* trigger file is the file containing the
*/

#include "tcplib.h"
#include <sys/time.h>
#include <stdlib.h>

#define POS_FILE 2
#define POS_MINDELAY 4
#define POS_MAXDELAY 5
#define POS_IP 3
#define POS_TRIGGER 1

//make code cleaner by shortenting cast
#define ip(p) ((TcplibIPPacket *)p)

void displayHelp();

int main(int argc, char ** argv) {
//check to see if user requested help
for (int i = 1; i < argc; i++) {
if ((strcmp(argv[i], "-h") == 0) ||(strcmp(argv[i], "--help") == 0)) {
displayHelp();
return 0;
}
}

if (argc != 6) {
displayHelp();
return 1;
} else {
uint32_t sourceip;
int mindelay = atoi(argv[POS_MINDELAY]);
int delayvar = atoi(argv[POS_MAXDELAY]) - mindelay;
int delay;
sourceip = strToIP(argv[POS_IP]);
if ((delayvar == 0) || (mindelay ==0)) {
displayHelp();
```

```
return 1;
} else {
struct timeval timestamp;
gettimeofday(&timestamp, NULL);
srand(timestamp.tv_usec);
TcplibFileReader * trigger = new TcplibFileReader(argv[POS_TRIGGER]);
TcplibFileWriter * writer = new TcplibFileWriter(argv[POS_FILE], trigger->getFileHeader());
TcplibPacket * p;
while (!trigger->eof()) {
p = trigger->getNextPacket();
if(p->isICMPPacket()) {
if(((TcplibICMPPacket *)p)->getICMPType() == ICMP_ECHO) {
ip(p)->setIPDestAddress(ip(p)->getIPSourceAddress());
ip(p)->setIPSourceAddress(sourceip);
((TcplibICMPPacket *)p)->setICMPType(ICMP_ECHOREPLY);
ip(p)->setIPChecksum(ip(p)->calculateIPChecksum());
timestamp = p->getTimestamp();
delay = mindelay + (rand() % delayvar);
timestamp.tv_usec += (delay);
while (timestamp.tv_usec >= 1000000) {
timestamp.tv_sec++;
timestamp.tv_usec -= 1000000;
}
p->setTimestamp(timestamp);
ip(p)->setIPTTL(64);
((TcplibICMPPacket *)p)->setICMPChecksum(((TcplibICMPPacket *)p)->calculateICMPChecksum());
ip(p)->setIPChecksum(ip(p)->calculateIPChecksum());
writer->writePacket(p);
}
}
delete p;
}
delete writer;
return 0;
}
}
}

void displayHelp() {
cout << "simSmurf2b:" << endl << endl
     << " usage: simSmurf2b <trigger file> <outfile> <host> <min delay> <max delay>" << endl;
}
```

B.3.4 simSmurf2.pl

```
#!/usr/bin/perl

# Jason McKenna, winter 2004

# Perl script for simulating that a network was the target to a smurf attack.
#  This script makes used of the simSmurf2 program , built on TCPLib.


# modified 11 Feb 2004 JT
# modified 15 Feb 2004 JM

# tcpjitter args
$jittersource = "d1 t30"; #jitter on incomming packets
$jittereach = "d0.5"; #drop 0.5% of reply packets (due to large net traffic
$jitterfinal = "t20"; #jitter timestamps 20% AFTER merging.

#hash of attackers
# each hash represents the suffix of the IP for the host responding, and the
```

```
# min delay for response.  Max delay is (2)(min delay)
%att = (
"1" => 141,
"2" => 131,
"3" => 134,
"8" => 342,
"10" => 112,
"11" => 105,
"12" => 98,
"13" => 93,
"14" => 105,
"15" => 107,
"20" => 214,
"21" => 250,
"23" => 204,
"24" => 199,
"25" => 201,
"26" => 157,
"28" => 293,
"29" => 261,
"30" => 220,
"31" => 278,
"33" => 134,
"45" => 203,
"46" => 210,
"47" => 221,
"48" => 205,
"49" => 213,
"53" => 167,
"66" => 183,
"71" => 203,
"72" => 223,
"73" => 101,
"80" => 302,
"81" => 289,
"82" => 319,
"83" => 401,
"88" => 134,
"89" => 154,
"112" => 222,
"113" => 230,
"121" => 223,
"132" => 303,
"144" => 132,
"145" => 128,
"146" => 191,
"149" => 178,
"151" => 164,
"152" => 129,
"153" => 107,
"155" => 184,
"182" => 223,
"192" => 334,
"193" => 245,
"194" => 267,
"203" => 112,
"205" => 343,
"232" => 349,
"237" => 393,
"244" => 159,
"245" => 161,
"251" => 402);

#create trigger file
```

```
system("./simSmurf2a smurf2part1-orig.tcp 10000 100000");
system("tcpjitter smurf2part1-orig.tcp smurf2part1.tcp $jittersource");
system("rm smurf2Part1-orig.tcp");
$files = "smurf2part1.tcp";

$ournet = "172.168.199.";

foreach $host(keys(%att)) {
print ("IP $host delaying for " . $att{$host} ." us\n");
system "./simSmurf2b smurf2part1.tcp smurf2part$ournet$host" . "a.tcp $ournet$host $att{$host} " . (2 * $att{$host});
#print "./simSmurf2b smurf2part1.tcp smurf2part$host" . "a.tcp $host $att{$host} " . (2 * $att{$host}) . "\n";
system("tcpjitter smurf2part$ournet$host" . "a.tcp smurf2part$ournet$host\.tcp $jittereach");
system("rm smurf2part$ournet$host" . "a.tcp");
$files = $files . " smurf2part$ournet$host\.tcp";
}

print "Merging...\n";
$mergecmd = "tcpmerge $files smurf2.tcp";
print "$mergecmd \n";
system "$mergecmd";

print "Cleaning up...\n";

print "rm $files\n";
system ("rm $files");

print "Done\n";
```

B.3.5 simSmurf3b.cc

```
/* Copyright
* (C) Her Majesty the Queen, as represented by the Minister of National Defence,
* 2003
*
* (C) Sa majeste la reine, representee par le ministre de la Defense nationale,
* 2003
*
* Written by Jason McKenna, summer 2003 for DRDC
*/

/* simSmurf3b.cc
* This will attempt to simulate part of a smurf attack.  Imagine the situation
* where a network has been pinged to the broadcast address.  Each computer in
* on the network will respond to the ping.  Now if the source address in the
* original IP header had been spoofed, then the victim (the machine with the
* spoofed address) is flooded.  This program will simulate the response from
* a single host on the network which has been pinged.  It will read in the
* timestamps from the specified tcpdump input file (the packets do not need to
* be ICMP echo requests, all we are using them for is a timestamp), delay the
* specified number of microseconds, and send a response based around the packet
* format from the original smurf.c.
*/
#include "tcplib.h"
//This is used to define the length of the text of the echo message
#define ECHO_MSG_LEN 190

//This MAC was associated with the address above from the 2002 traffic.
// This is just the MAC of the next link in the chain connecting to the host
#define SOURCEMAC { 0x00, 0x08, 0xe3, 0x17, 0xd0, 0x90 }

// MAC of the router
#define DESTMAC { 0x00, 0x30, 0x80, 0xce, 0xba, 0xa2 }
```

```
#define OS_WIN 0
#define OS_LINUX 1

//marker used to specify that a field should be randomized
#define RAND -65536
//Describe how Windows machines reply to pings
#define WIN_IP_ID_INIT RAND
#define WIN_IP_ID_DELTA 256
#define WIN_TTL 122 //initially set to 128, but there are some hops

//descrive how 'nix reply to pings
#define LINUX_IP_ID_INIT RAND
#define LINUX_IP_ID_DELTA 1
#define LINUX_TTL 249 //initiall set to 255

#define POS_INFILE 1
#define POS_OUTFILE 2
#define POS_SOURCEIP 3
#define POS_TARGETIP 4
#define POS_DELAY 5
#define POS_OS 6

void displayHelp();

int main(int argc, char **argv) {
for (int i = 1; i < argc; i++) {
if ((strcmp(argv[i], "-h") == 0) || (strcmp(argv[i], "--help") == 0)) {
displayHelp();
return 0;
}
}
if ((argc != 6) && (argc != 7)) {
displayHelp();
return 1;
} else {
//open the file for reading
TcplibFileReader * reader = new TcplibFileReader(argv[POS_INFILE]);
//Generate output file header
pcap_file_header fh;
fh.magic = TCPDUMP_MAGIC; //tcpdump magic number
fh.pcap_version_major = 2; //major pcap version
fh.pcap_version_minor = 4; //minor pcap version
fh.thiszone = fh.sigfigs = 0;
fh.snaplen = 68; //default 68 byte capture length
fh.linktype = 1; //ethernet
//this is what we'll be using to write to the file
TcplibFileWriter * writer = new TcplibFileWriter(argv[POS_OUTFILE], fh);

TcplibICMPPacket * p = new TcplibICMPPacket(fh.snaplen);

//modify the packet header
pcap_packet_header ph = p->getRawHeader();
ph.len = ETHER_HDR_LEN + sizeof(iphdr) + sizeof(icmphdr) + ECHO_MSG_LEN;
p->setRawHeader(ph);

p->setIPPacketLength(sizeof(iphdr) + sizeof(icmphdr) + ECHO_MSG_LEN);

int delay = atoi(argv[POS_DELAY]);

//generate ethernet header
//generate random MACs
uint8_t smac[ETHER_ADDR_LEN] = SOURCEMAC;
uint8_t dmac[ETHER_ADDR_LEN] = DESTMAC;
//set MACs & protocol type
```

```
p->setEthernetSourceMAC(smac);
p->setEthernetDestMAC(dmac);

//set up the IP header
p->setIPVersion(IPVERSION);
p->setIPHeaderLength(5);
p->setIPSourceAddress(strToIP(argv[POS_SOURCEIP])); //make the apparent target of the smurf

p->setIPDestAddress(strToIP(argv[POS_TARGETIP])); //send packet to broadcast address
int os;
if (argc == 7) {
if (strcmp(argv[POS_OS],"-w") == 0) {
os = OS_WIN;
} else if (strcmp(argv[POS_OS],"-l") == 0) {
os = OS_LINUX;
} else {
displayHelp();
return 1;
}
} else {
os = OS_WIN;
}

int ttl;
int ipid;
int iddelta;
switch (os) {
case OS_WIN:
ttl = WIN_TTL;
ipid = WIN_IP_ID_INIT;
iddelta = WIN_IP_ID_DELTA;
break;
case OS_LINUX:
ttl = LINUX_TTL;
ipid = LINUX_IP_ID_INIT;
iddelta = LINUX_IP_ID_DELTA;
break;
default:
cout << "ERROR: This message should never appear." << endl;
return 1;
}
if (ipid == RAND) {
ipid = rand() % 65536;
}
p->setIPDontFragBit(true); //this should be the only IP flag set
p->setICMPType(ICMP_ECHOREPLY);

TcplibPacket * pt;
struct timeval ts;
while (!reader->eof()) {
if (ttl == RAND) {
p->setIPTTL(rand() % 256);
} else {
p->setIPTTL(ttl);
}
pt = reader->getNextPacket();
ts = pt->getTimestamp();
ts.tv_usec += delay;
if (ts.tv_usec > 1000000) {
ts.tv_sec += (ts.tv_usec / 1000000);
ts.tv_usec %= 1000000;
}
p->setIPID(ipid);
p->setTimestamp(ts);
```

```
delete pt;
p->setIPChecksum(p->calculateIPChecksum());
p->setICMPChecksum(p->calculateICMPChecksum());
writer->writePacket(p);
if (iddelta == RAND) {
ipid += (rand() % 65536);
} else {
ipid += iddelta;
};
ipid %= 65536;
}
delete reader;
delete p;
delete writer;
return 0;
}
}

void displayHelp() {
cout << "simSmurf3b help" << endl << endl

     << "  Simulates the response from one computer on a network to a spoofed source IP." << endl
     << "    Used in conjunction with simSmurf3a.pl to simulate entire smurf attack." << endl << endl

     << "  usage: simSmurf3b <input timestamp file> <output file> <source IP> <target IP> <delay> [os]" << endl << endl

     << "    <input timestamp file> - TCPDump file containing packets with timestamps to treat as pings" << endl
     << "    <output file> - Name of file to write to." << endl
     << "    <source IP> - IP address of computer sending ping responses" << endl
     << "    <target IP> - IP of computer being smurfed" << endl
     << "    <delay> - Delay of packets due to network noise (us)" << endl
     << "    [os] - (optional) OS to mimic (-w = Windows, -l = GNU/Linux)" << endl << endl;
}
```

B.3.6 simSmurf3a.pl

```
#!/usr/bin/perl

# Perl script for simulating that a network was the target to a smurf attack.
#  This script makes used of the simSmurf3b program, built with TCPLib.
#  In order to simulate a number of hosts responding to a particular PING
#  request, we create a file which has a number of packets.  This file is used
#  to synchronise the responses, leading to this impression that all hosts are
#  responding to the same packet.

#constants (make these command line options in next version?)

$target = "172.168.244.255";
$attackerPrefix = "61.182.0."; #netscan.org reports that this subnet responds w/
#69 responses to a ping at 61.182.0.255, but I didn't actually send a
#ping so the unique IPs and OSs below are just hypotheticals

$delay = 240;

#ugly but it works (note to self -- use a hash next time)
@attackers = (1 , "l", 2 ,"w", 3 ,"w", 4 ,"l", 8 ,"l", 9 ,"w", 10 ,"l", 11 ,"w" ,
12 ,"w", 13 ,"w", 17 ,"w", 18 ,"l", 20 ,"w", 21 ,"w", 22 ,"w", 23 ,"l", 24 ,"l" ,
25 ,"w", 27 ,"w", 28 ,"w", 29 ,"w", 31 ,"w", 32 ,"l", 33 ,"l", 35 ,"w", 38 ,"w" ,
128 ,"l", 129 ,"w", 130 ,"w", 131 ,"w", 132 ,"l", 134 ,"w", 135 ,"l", 152 ,"w", 158 ,"l" ,
159 ,"w", 162 ,"l", 163 ,"l", 164 ,"l", 165 ,"w", 166 ,"l", 167 ,"w", 172 ,"w", 173 ,"w" ,
174 ,"w", 175 ,"l", 176 ,"w", 177 ,"w", 178 ,"w", 179 ,"l", 180 ,"w", 181 ,"w", 182 ,"w" ,
188 ,"w", 189 ,"w", 192 ,"w", 204 ,"w", 220 ,"w", 221 ,"w", 222 ,"w", 238 ,"l", 242 ,"w");
```

```
#rate of packets (per second, 10 is a good number)
$rate = 10;

#amount of time to simulate trace for (seconds)
# max is (2^16 - 2 (=65534?))/$rate
# another way of looking at this is "how many pings to broadcast address"
$simtime = 3000; # if #rate = 10, this is 300s or 5 mins

#max amount to shift trace by (us)
$maxshift = 500; # = 0.5ms

#tcpjitter args
$jitter = "t20 d1"; #jitter time by up to 20%, drop 1% of packets

#name of timer file (will be deleted, so be sure it's original
$timerfile = "timer.tcp";

#command to generate timer file
$gentimer = "./simPortScan $timerfile 1.1.1.1 2.2.2.2 1 $simtime $rate";

#ok, now we do stuff
print "Generating timer file...\n";
system "$gentimer";
print "\nCreateing traces...\n";
$mergecmd = "tcpmerge";
$n = 0;
foreach $host(@attackers) {
if ($host ne "l" && $host ne "w") {
print "From host $attackerPrefix$host... simulating...\n";
$cmd = "./simSmurf3b $timerfile smurf3.$host.tcp $attackerPrefix$host $target $delay -";
$cmd = $cmd . @attackers[$n + 1];
$n = $n + 2;

system "$cmd";
$r =rand($maxshift);

$r =~ /\./;
$r = $' + 1;
print "Making more realistic...\n";
system "tcptimeshift smurf3.$host.tcp smurf3.$host.a.tcp 0 $r";
system "tcpjitter smurf3.$host.a.tcp smurf3.$host.tcp $jitter";
$mergecmd = $mergecmd . " smurf3.$host.tcp";
print "Cleaning temp files...\n";
system "rm smurf3.$host.a.tcp";
}
}

print "Merging...\n";
$mergecmd = $mergecmd . " smurf3.tcp";
system "$mergecmd";

print "Cleaning up...\n";
foreach $host(@attackers) {
if ($host ne "l" && $host ne "w") {
system "rm smurf3.$host.tcp";
}
}
system "rm $timerfile";
print "Done\n";
```

B.4 TFN2K DoS Generator

B.4.1 even.pl

```
#!/usr/bin/perl

# tfnx are tfn2k-syn-x timeshifted to same timespace

#apply TTL modifiers:
# client 1: 5 hops
# client 2: 12 hops
# client 3: 17 hops
# client 4: 8 hops
# client 5: 20 hops

system "tcpttltranslate tfn1.tcp tfn1-2.tcp -5";
system "tcpttltranslate tfn2.tcp tfn2-2.tcp -12";
system "tcpttltranslate tfn3.tcp tfn3-2.tcp -17";
system "tcpttltranslate tfn4.tcp tfn4-2.tcp -8";
system "tcpttltranslate tfn5.tcp tfn5-2.tcp -20";

#merge tfns and stretch them out to just above 5Mbps
system "tcpmerge tfn1-2.tcp tfn2-2.tcp tfn3-2.tcp tfn4-2.tcp tfn5-2.tcp tfn_a.tcp";
system "rm tfn1-2.tcp tfn2-2.tcp tfn3-2.tcp tfn4-2.tcp tfn5-2.tcp";
system "tcptimestretch tfn_a.tcp tfn_b.tcp 5.7";
system "rm tfn_a.tcp";

#extract heart of DDoS (remove any delay from starts)
system "tcpslice -w tfn_c.tcp +2 +107 tfn_b.tcp";
system "rm tfn_b.tcp";

#make trace 68 byte packet cap len
system "tcptrunc tfn_c.tcp tfn_d.tcp 68";
system "rm tfn_c.tcp";

#clean traffic is divided into part1.tcp part2.tcp and part3.tcp.  part2 is
# length of tfn_d.tcp

#align in time and merge two traces for part 2
system "tcptimeshift tfn_d.tcp tfn_e.tcp -109569783 -854176";
system "rm tfn_d.tcp";
system "tcpmerge part2.tcp tfn_e.tcp part2a.tcp";


for ($i = 0; $i < 106; $i++) {
#extract the current second from the file
system "tcpslice -w part2_$i.tcp +$i +1 part2a.tcp";
system "tcpslice -w ddos2_$i.tcp +$i +1 tfn_e.tcp";
system "tcpslice -w clean_$i.tcp +$i +1 part2.tcp";
#find the rate in the current second
system "tcprate part2_$i.tcp > temp.txt";
open FH, "temp.txt";
$f = <FH>;
$f =~ /\s/;
$rate = $';
$f = <FH>;
$f =~ /\s/;
$rate += $';
print "$rate\n";
if ($rate > 579570) {
$float = ($rate - 579570)/$rate * 100;
print "Dropping $float %\n";
system "tcpjitter ddos2_$i.tcp ddos3_$i.tcp d$float";
system "tcpjitter clean_$i.tcp clean3_$i.tcp d$float";
```

```
} else {
system "cp ddos2_$i.tcp ddos3_$i.tcp";
system "cp clean_$i.tcp clean3_$i.tcp";
}
$mergeclean = $mergeclean . " clean3_$i.tcp";
$mergeddos = $mergeddos . " ddos3_$i.tcp";
system "rm part2_$i.tcp ddos2_$i.tcp clean_$i.tcp";
close FH;
}
system "rm part2a.tcp tfn_e.tcp";

print "tcpmerge $mergeclean part2_clean.tcp\n";
system "tcpmerge $mergeclean part2_clean.tcp";
print "tcpmerge $mergeddos part2_ddos.tcp\n";
system "tcpmerge $mergeddos part2_ddos.tcp";
print "tcpmerge part2_ddos.tcp part2_clean.tcp part2_mix.tcp\n";
system "tcpmerge part2_ddos.tcp part2_clean.tcp part2_mix.tcp";

system "rm $mergeclean $mergeddos temp.txt";
```

Annex C
List of Acronyms

ACK/RST/SYN/FIN	TCP header flags
ASCII	American Standard Code for Information Interchange
DDoS	Distributed Denial of Service
DEFCON	DEFense CONdition
DoS	Denial of Service
DREnet	Defence Research Establishment network
DRDC	Defence Research and Development Canada
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IP ID	Internet Protocol IDentification header field
NetBIOS	Network Basic Input/Output System
OS	Operating System
TCP	Transmission Control Protocol
TTL	Time To Live
UDP	User Datagram Protocol
VA	Vulnerability Assessment

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence R&D Canada – Ottawa
3701 Carling Ave., Ottawa, ON K1A 0Z4

2. SECURITY CLASSIFICATION (overall security classification of the document, including special warning terms if applicable)

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C or U) in parentheses after the title.)

Network Attack Reference Data Set (U)

4. AUTHORS (Last name, first name, middle initial)

McKenna, J. and Treurniet, J.

5. DATE OF PUBLICATION (month and year of publication of document)

December 2004

6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc.)

72

6b. NO. OF REFS (total cited in document)

33

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address.)

Defence R&D Canada – Ottawa
3701 Carling Ave., Ottawa, ON K1A 0Z4

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant)

15bf29

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written)

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.)

DRDC Ottawa TM 2004-242

10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor)

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(x) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

Full unlimited announcement

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

A set of network attacks was created at DRDC Ottawa for the purpose of testing network attack detection and visualisation methods. The network attack traces were generated by extracting attacks from real-world networks, from closed networks specifically set up to test attacks, and through the use of custom software written to simulate attack traffic. In this document, the attacks included in the data set are described in detail along with the method used to generate them. The software tools used in the creation of the data sets are presented and issues involved in the generation of the data are discussed. The 52 attack traces are available on a CD in a purified form.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Network attack
Traffic analysis
Network traffic
Scans
Network reconnaissance
Denial of service

Defence R&D Canada

Canada's leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca